The application of proof plans to computer configuration problems

Helen Lowe

Ph.D.

University of Edinburgh

1993

Abstract

Traditional expert systems technology is limited, being hard to maintain and extend to new problems. In this thesis, I propose a logical formalization for the domain of computer hardware which will enable the use of theorem proving techniques for the task of computer hardware configuration. This domain was the subject one of the earliest knowledge based systems, XCON. Whilst XCON is cited as a successful system, it has nevertheless also been criticized for its maintenance problems. This is a important issue, as the turnover of computer hardware components is particularly changeable, the market being subject to intense competition and rapidly changing technology. My approach enables the task of configuring a computer configuration c from a specification $spec(c)$ to be performed by synthesizing c as a by-product of proving the theorem

$$\exists c.spec(c)$$

when c becomes instantiated in the course of the proof. A clean separation of the object-level, heuristic, and control knowledge enables us to guide search and aids maintenance. As well as ensuring legal configurations, by virtue of the soundness of the underlying logical theory, I have also been able to take design issues into consideration by using heuristic and control knowledge.

Acknowledgements

I am indebted to many people. First and foremost, to Alan Bundy, who supervised me throughout this project, and provided both inspiration and much practical help. Frank van Harmelen was one of my supervisors both during the first stage of my PhD work and also for my M.Sc. project, when, amongst other things, he taught me how to write clear technical prose, for which I shall always be grateful. Geraint Wiggins was particularly helpful during the later stages, and added much precision to the final document. Several people at Hewlett Packard helped me gather the extensive domain knowledge necessary, particularly Janet Bruten, Richard Hull, Alison Kidd, John McShane, Stefek Zaba, Steven Owen, Chris Preist, and John Hanahan. Many people in the Mathematical Reasoning Group gave invaluable feedback, stimulation, friendship, and support throughout this project; I should like to mention in particular Andrew Ireland, Jane Hesketh, Alan Smaill, Paul Brna, and David Basin.

Lastly, I should like to express my gratitude to my husband, John, for reading every word and squiggle, for helping me with the technicalities: proof-reading, photocopying, and binding, and for his general support and encouragement.

This thesis is dedicated to my children, and to the memory of my parents.

Table of Contents

List of Figures

Chapter 1

Introduction

"The criterion of the scientific status of a theory is its falsifiability, or refutability, or testability."

Karl Popper

1.1 Automated reasoning

The subject of this thesis is the exploration of a particular approach to automated reasoning applied to a novel domain. The approach is that of *proof planning*, initially applied by Bundy *et al.* (1991b) to the domain of proving theorems by mathematical induction. I wanted to tackle a significantly different, non-mathematical domain using the above basic approach, in order to investigate the hypothesis in Bundy (1988) that the proof planning technique could be expected to have

> "application to any area of automated reasoning where search control is a problem ... to guide the reasoning of expert systems." (Bundy, 1988)

The chosen domain is that of computer hardware configuration, where search control is one of the main problems and where strategies to control it are much sought after.

Planning as a general activity is the making of an orderly sequence of action that will lead to the achievement of a stated goal or goals (Hall, 1975, page 6).

A *proof plan* is a means of expressing the commonality between members of the same "family" of proofs (Bundy, 1991), whilst allowing sufficient flexibility and adaptability to prove a large number of different theorems. Proof plans provide an expression of strategies for automatic reasoning by describing *tactics*, which can be understood for now as a kind of operation (or program), in terms of the *preconditions* under which they are *applicable*, and their *effects* if successfully applied. The specification of a tactic in terms of its preconditions and effects is called the *method* for that tactic, and methods provide a basis for combining tactics (the effects of one implying the preconditions of subsequent methods) to form a complete plan which, if executed, will carry out a reasoning task, such as proving a theorem.

There are various motivations for seeking to automate a task normally or exclusively performed hitherto by human agents. Apart from the obvious ones of saving effort, or cutting costs, a common AI motivation is that it is intrinsically interesting to discover whether the elusive processes of human reasoning can be understood enough to allow simulation by program. Such an understanding of the reasoning behind human thought and action might greatly facilitate the construction of an automated system which is both powerful and malleable; which can stand up to being tested exhaustively, and by the most tricky of cases; which can be adapted to similar yet different problems; and which can, if it is so wished, produce output transparent enough for human consumption.

From this standpoint, the following systems would not be judged successful, even though they performed well on a well-defined set of tasks:

- A program which is not readily maintainable by reason of its over-sensitivity to small changes — one addition to the objects known about by the system necessitating large changes in the whole "reasoning" process. This fails because of its *lack of knowledge structure.*

- A theorem prover whose sequence of proof steps is globally incomprehensible to the human observer, beyond being able to verify that each step follows legally from the last — lacking any sense of an overall motivation for the

proof at any higher level, other than the fact that it "succeeds" in the end. This fails because of *inscrutability.*

- A theorem prover which proves a certain corpus of theorems but which cannot prove others even of a similar form, and which cannot be conceived of as ever being able to prove such theorems. This fails through *lack of generality.*

- A system which, in order to solve more than a small subset of problems, incorporates increasingly *ad hoc* techniques.

Let us turn now to the domain of interest, before considering why a technique developed for the formalized, well-defined world of mathematics is considered suitable for the rather more down-to-earth, pragmatic, informal world of computer hardware configuration.

1.2 Hardware configuration

1.2.1 Problems with earlier approaches

There are many configuration systems around: every major manufacturer of hardware has one[1] my prototype and it is an interesting domain for the general researcher to test ideas on. Automated configuration has a long history in AI terms: one of the earliest, most successful knowledge based systems was built to tackle the problem of configuration (McDermott, 1982). The motivation was simply good business sense; a task which was difficult and where mistakes were frequently made but which seemed essentially a matter of remembering and following rules was a good candidate for automation. The knowledge of the experts was canned and made available to the inexpert and ignorant.

---

[1]In the context of my test domain (HP systems) see, for example, Merry (1992), although note that this system has a different orientation from mine: it is a point of sales support system for configuration.

However, several problems emerged. I shall not dwell on these here, since they are more properly the concern of Chapter 2, but, briefly, they were:

- It was hard to maintain the system because facts, heuristics, control knowledge, and context information were inextricably intermixed.
- The rule-based system was not based on a firm logical foundation.
- Revolutionary changes in technology passed each system by. Automated systems for configuring more up-to-date hardware are fundamentally different from their predecessors. There is no reason to suppose that they will not become obsolete in their turn.

Let us examine how my approach differs from this.

1.2.2 Separation of knowledge

In many tasks — not just configuration and theorem-proving — we observe that the "knowledge" is of different kinds.

1. Factual:
   - The sum of the successor of an integer x and an integer y is equal to the successor of the sum of x and y;
   - Every integer has a successor;
   - The total capacity of a string of disk drives is equal to the sum of the capacities of each disk drive in the string;
   - Every device in a configuration has a means of connection in the configuration.
2. Heuristic:
   - Evaluation functions for heuristic search (*e.g.* for hill-climbing)

- Knowledge about potentially incompatible devices in configuration. If devices are known to be incompatible (under any circumstances) this "heuristic" may be elevated to a "fact"; often there is no such certainty, and we retain the knowledge as a heuristic.

3. Control:

   - A proof plan for induction: choose variable to induce on, split into base and step case(s).
   - Straightforward cases of configuration: choose processor, then, ..., then configure disk drives, then assess backup needs,...

Referring to the problems mentioned in Section 1.2.1 above:

- In order to be expressible in logic, a system needs to be based on such a separation. This separation enables each type of knowledge to be encoded *declaratively* if we so wish. This is of the utmost importance, if we are to be able to check that the logical formalism given accords with our understanding of the semantics. We need to be able to check, separately and independently:

  - That the facts represented are "true", or at least what we intend (*e.g.* the axioms of a mathematical theory; the rules of configuration).
  - That procedures are captured correctly.

  If these two are inextricably intermixed, the configuration task becomes unacceptably hard.

- A system based on a clean separation is easier to maintain, because knowledge is encoded declaratively. Hayes (1977) gives a defence of the use of logic to avoid precisely this confusion, and this case is argued further both in this thesis and in Lowe (1991b).

- It is more likely that we will be able to salvage *something* in the face of technological innovation, provided it is not too extreme. For example, if there is a radical change in storage methods and components, then it will not affect the top-level architecture. Methods of generating partial configurations which are not affected by the changes to affected components will remain unscathed also.

1.2.3 Logical formalism

At first sight, it seems very difficult to formalize a domain such as configuration. Sales representatives "pick it up". Manuals don't tell you everything. Experience is the best teacher. And so on.

But sales representatives make mistakes. Manuals would be the better for being complete and unambiguous. Experienced sales representatives do not necessarily make good teachers and the mistakes of inexperienced ones cost money. Generally speaking, expert systems, supposedly based on the distilled knowledge of experts, are *not reliable* (Bundy, 1987b).

Merely because it is difficult, formalizing the computer hardware domain is an important and useful task. If it is done in a methodical way, by capturing knowledge, testing this knowledge in its captured form, weeding out errors of transcription, of understanding, and those due to ambiguity, then we end up with a robust system which *cannot* produce erroneous results.

At first, it seems that capturing expert knowledge is too difficult. The expert can take short cuts; some knowledge appears to be "hardwired" and is used automatically and non-introspectively when it is appropriate; analogical reasoning, and adapting last week's order all play their part in the way the expert carries out her task. Does the mathematician know why the theorem was proved this way? This kind of "inspiration" does not appear at all amenable to automation.

However, we can capture a fair proportion of this kind of knowledge. Let us look at the teaching of mathematics first by way of example. It is now recognized that teaching methods should be based on a greater understanding: for example, in teaching subtraction the method of decomposition has taken over from the old "borrow-and-pay-back" ritual. This is seen as being better than attempting to "hardwire" knowledge into children by giving them an example to look at, and then twenty others which are similar to the example to do themselves.

At a slightly higher level, the other side of the coin to "inspiration" is "confusion". Consider the equation solving system PRESS (Bundy & Sterling, 1981). This imposes structure on the amorphous mass of rewrite rules so often applied

in desperation, without motivation or appropriateness, by examination-panicked students. Instead of searching blindly through all the object-level rules, of which there are a very great number, PRESS divides them into classes.

For example:

- *isolation*: rewrite rules which are applicable if there is single occurrence of the variable, yielding this variable alone on the left hand side, *e.g.*

  $$x + a = b \Rightarrow x = b - a$$

- *collection*: combining multiple occurrences of the variable, *e.g.*

  $$a.x + b.x \Rightarrow (a + b).x$$

- *attraction*: moving occurrences of the variables closer together in the tree, *e.g.*

  $$log_b(x) + log_b(y) \Rightarrow log_b(x \times y)$$

  as shown in Figure 1–1

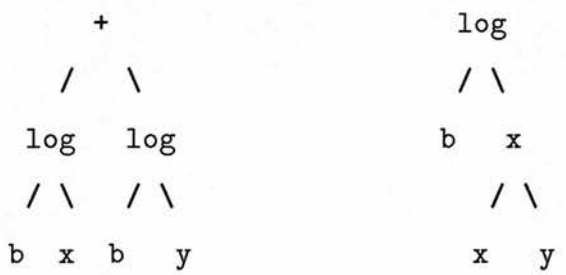

Figure 1–1: Attraction: moving the variables x and y closer together

— even the names are suggestive of the motivation and relative appropriateness of these rules to the current step in the solution. Here we have an automated reasoning system which has the potential to augment the explanatory powers of the teacher and provides a few pegs for the student to hang strategic *aides-memoire*.

Turning to the configuration domain, there are cases where lack of understanding of the *reasons* behind a rule are dangerous. Rules are expressed declaratively in the form

$$hypothesis \rightarrow conclusion$$

but may be implemented procedurally as

$$condition \rightarrow action$$

However, they may simply be understood as

$$[] \rightarrow action.$$

If this is what is believed, then the so-called human expert will not even check the hypotheses (being unaware of them) before reaching the conclusion (not being aware that it *is* a conclusion from hypotheses), and an undesirable or at best unnecessary action may result.

What of adaptation (made respectable in AI as *case-based reasoning*) and analogy? Although I do not consider this approach here, analogy, properly used, is a useful tool and can lead us to valuable insights. However, adapting a configuration produced in answer to last week's specification to meet the needs of this week's may not give the best results. Again, the reasons behind the choices made may have been lost and only the choices themselves remain — given access to the reasoning processes, I believe there is a case for utilizing this approach, but I also believe that this too has potential for automation. Chapter 8, Section 8.2.3 suggests how a similar task (configuration upgrading) could be tackled.

1.2.4 Progress: building on the past

I do not claim that it is possible to build a system which will configure everything from washing machines to works of art. However, it would be good to establish some generalities about configuring computer hardware systems: this is at least a desirable, as well as a realizable aim. For example:

- There are certain things which form an intrinsic part of *all* computer systems for the foreseeable future: processors, memory, storage of some kind, devices for communicating, and for producing hard copy. They may, of course, look very different in the year 2050 from what they do now.

- All devices have to be connected up — somehow. Ever faster, more compact, more versatile ways of doing so will, no doubt, be found in the future.

In order to achieve the highest level of generality, the greater the degree of abstraction possible the better. Where I can talk about "storage devices", rather than about disk drives, tape drives, *etc.*, I do so. If it is appropriate, I shall use abstractions rather than specific names — "connection" rather than "RS-232 cable", "modem", *etc.*

In re-appraising past theories and current systems, I have attempted to absorb the knowledge they give at the most general level possible, so as to be better able to draw out what is useful for my purposes, and to try and ensure that what I do has the widest applicability possible, within reason.

1.3 Proving theorems: configuring computers

1.3.1 Capturing higher level reasoning

I now turn, as promised, to the question of how a technique developed for use in mathematics can be useful in another domain.

First of all, it is a feature of human attitudes to problem-solving that we tend, at least in the first instance, to use strategies which worked for us before. So when faced with a new problem, it is natural to ask "Does this remind me of anything? How is it similar? How is it different? What did I do last time? Will it work? If not, what is different?" and so on. This is ably presented in Polya's "How to solve it" list, from the book of the same name (Polya, 1945).

When learning about configuration, as practiced and explained by experts, it was encouraging to find that innumerably different problems can be fitted into one or other of a small number of "templates" (see Chapter 3, Section 3.3.2, and Figures 3–5&3–6), and a large search space is divided and partly conquered by this strategy.

By using proof planning to guide inference, I aimed to separate domain knowledge and control information. By separating the task into appropriate levels, I can then reason about aspects of the task at the right level of detail. Decisions can be left until all the knowledge necessary to make an informed choice has been gathered. I can also explicitly represent strategies for directing the search.

1.3.2 Benefits of proof planning

Proof planning techniques can benefit the development and maintenance of knowledge based systems based on firm logical foundations in many ways.

Firstly, using planning techniques can be advantageous in itself, provided the planning overhead does not outweigh the gains. Planning may pay off for two reasons:

1. Planning may be more efficient than execution, in that the steps involved in planning an operation may be fewer, or less complicated, than those involved in carrying it out.

2. The planning process is guided by the preconditions of methods (see page 1), and if these preconditions are sufficiently strong, search should be much more directed. Instead of hundreds of choices at every point there should be only a few.

Secondly, if there is a clean separation of the object-level theory from meta-level knowledge, it is then possible to consider a number of related problems, all using the same underlying object-level theory. For example, connecting up a device is the same whether both device and connectors are being synthesized (as when the

top-level problem is to find a configuration to meet a specification) or whether they appear in a customer's order (as in the case where the problem is to check such an order drawn up by a sales representative).

Thirdly, there are potential benefits for interactive or co-operative problem-solving systems. If a plan fails, we want to *understand* why. Using methods as building blocks, the failure is returned at the appropriate level. In fact the analogy is that of a block which is actually a preassembled compound put together off-site to fulfill a particular need.

And finally, as mentioned earlier, the object theory can be maintained to take account of new or obsolete components and other changes, separately from the meta-level knowledge. We can, at various times, make changes to the object-level theory, add new heuristics known or thought to be useful, acquire recently learned strategies, or experiment with strategies.

1.3.3 Configuration *via* synthesis

Having an intuition that configuration can be performed using techniques "similar" to the proof planning utilized in theorem proving and program synthesis is not, in itself, enough. It is a hypothesis only, and in order to test it out I had to develop some more rigid framework. In this case, it proved possible to use theorem proving techniques directly by stating the problem as a theorem, once I had the logical mechanisms in place which allowed this. In this case, the theorem to be proved has the form *"there exists ..."*, and states, as a conjecture, that there is some configuration which meets the specification. Thus in proving the conjecture

$$\exists c.spec(c)$$

where $spec(c)$ is a specification for a configuration c, c itself is synthesized as a by-product of proving this theorem. This is explained fully in Chapter 4.

1.4 Methodology

1.4.1 Difficulties of formalizing the domain

One aim of this research was the development of a robust, maintainable automatic configurer. The success of this enterprise may be judged on

- Whether the solutions given by the automatic configurer correspond with solutions judged to be acceptable by human experts.
- Whether solutions can be found in a reasonable time.
- Whether better solutions are found in preference to less good ones: *i.e.* the better solutions are returned first.
- Whether the automatic configurer dealt with search in a desirable way; *e.g.* whether a request for further solutions would produce configurations which were acceptable, but also discernibly different from previous solutions in some significant way — *i.e.* using different components, and not merely trivial permutations of other solution configurations (see Chapter 7, Section 7.8.3).
- Whether any reasonable solution will, ultimately, be found.
- Whether the automatic configurer was maintainable, in a number of senses.

The question of maintenance, in all its senses, will be dealt with in the main part of this thesis, but for now let us concentrate on one aspect of this question: whether the system proposed is adaptable to changes and additions to the theory.

Computer configuration in many ways is an ill-defined task. Folklore might have it as an art rather than a science. In rejecting this view, I need to be able to describe and define certain concepts accurately. Later I shall define, for example, what a *legal* configuration is; what the *utility* of a system is; how components may

be described in terms of *usage*; what it means for a solution to be locally optimal — because I decided that these were useful concepts, indeed necessary ones, in drawing up a theory of configuration which could then be used as the basis for an automated system. The actual usefulness of these concepts and of all the axioms, rules, and heuristic knowledge assembled was tested by designing and building an automatic configurer based on them, and assessing its output — *viz* whether it gave results consistent with what I expected, and that corresponded to the output that experts would give.

1.4.2 Trial and Error

However cleanly presented in this thesis, the process of developing the ideas intimated above was not a straightforward one. Trial and error played a large part. And so a fairly strict methodology was followed, since scientific method is as important in AI as in any empirical science.

Popper (1989, pages 312–315) analyses the process of trial and error as one in which we are actively involved in trying to solve a definite problem, rather than simply (passively) observing phenomena. Theories are merely tentative hypotheses, to be tested critically; if they are at fault then this is to be taken in a positive spirit, so that they may be revised:

> "... we make progress if, and only if, we are prepared to *learn from our mistakes.*" (Popper, 1961, page 87)

We can never "prove" that a theory is true (in the sense that it is an accurate model of the world, or subject of study), although we may sometimes refute it. Our aim should be to subject our theories to the most stringent tests possible:

> "Only if we cannot falsify them in spite of our best efforts can we say that they have stood up to severe tests.... For if we are uncritical we shall always find what we want: we shall look for, and find, confirmations, and we shall look away from, and not see, whatever might be dangerous to our pet theories.... to ensure that that only the fittest theories survive, their struggle for life must be made severe for them." (Popper, 1961, pages 133–134)

1.4.3 Testability

Elsewhere in the thesis I deal with the amassing of knowledge, its division into categories (facts, heuristics, *etc.*), and its encoding; but what to collect, and how to use it, were at first the objects of speculation. The relevance and accuracy of this information were tested in a systematic way. For example, configurations can be divided into substructures and each of these tested independently; the object-level theory was tested for verisimilitude before the development of the meta-level planning architecture, and so on. Results given by the program which was eventually implemented were tested against those found by human experts where possible.

My prototype system performed well judged by its performance in the tests I had devised. However, the end result — which is not a fully definitive system, even if regarded as a prototype — lacked the power to deal with many cases. It is not intended to be seen as a "proven" system and it does not embody even all of the strategies for configuration known to me. The stringent tests needed for this belong to field testing and were beyond the scope of my thesis work.

However, what is important here is that I have pursued the following:

1. Setting up a problem to be solved — *i.e.* how to automate the configuration of hardware from specifications.

   > "A scientist, whether theorist or experimenter, puts forward statements, or systems of statements, and tests them step by step... by observation and experiment." (Popper, 1990, page 27)

2. Actively pursuing this task in the light of a tentative theory of configuration.

   > "Experiment is planned action in which every step is guided by theory. We do not stumble over our experiences, nor do we let them flow over us like a stream. Rather, we have to be active: we have to *'make'* our experiences." (Popper, 1990, page 280)

3. Testing this system as far as possible, and making it available for others to test.

> "...it must be possible for an empirical scientific system to be refuted by experience ... What characterizes the empirical method is its manner of exposing to falsification, in every conceivable way, the system to be tested." (Popper, 1990, pages 41–42)

I believe that I have set up a solid framework according to this methodology; the automatic configurer which I have implemented performs successfully on many examples and a number of different computer systems. Of course, further testing and further development can be done, by myself or by others. The latter is perhaps more important: other people may be better motivated to attempt to refute our theories than we ourselves are, and therefore provide the more stringent testing desirable in order that only the fittest theories might survive, and indeed to motivate the development of the fittest theory. The following assist this process:

- The separation of object-level and meta-level knowledge gives clarity to the theory.
- Both are encoded declaratively. The implementation of the program is in Prolog, which assists with this aim.
- The results may be reproduced: any person may use the code given in the appendices and verify the purported solutions to sample tests given in the text; she may also devise new tests and attempt to falsify the theory or its implementation.
- The theory is amenable to *disjointed incrementalism* (as outlined in the next section).

This is in marked contrast with systems where the implementation details are hidden and which are not, therefore, subject to verification by others.

1.4.4 Disjointed Incrementalism

One tool in developing the theory and automation of configuration deserves a brief description here.

The technique of proof planning lends itself to the disjointed incrementalist approach borrowed from planning theory (Alden & Morgan, 1974). This is often known, somewhat ironically, as "the strategy of muddling through": a first stab at a theory, program, *etc.* is tested. The results of such tests (if not wholly successful) will show up weaknesses. With suitable analysis, the nature and pattern of failure will usually suggest modifications to the theory, which it is hoped will improve it. For a theorem prover, failure might mean the inability to prove a particular theorem. If the failure is analysed it may suggest what was deficient about the theorem prover in its initial form and what could be changed to enable it to prove the theorem in question. Changes are made: obviously we must retest all the cases which were successful before, to check that they still are; and we test the failures also. Progress means that more cases are successful than previously; if the changes introduce more problems than they solve they may be abandoned; otherwise the process repeats. In theory, this process never ends, since it is always conceivable that a new test, or a new phenomenon, may introduce failure, but in practice testing may be eventually suspended. Disjointed incrementalism is a pragmatic approach, in that, if a target is set, and a given theory or system meets this target, it will not be further amended unless the target is reset. The following is a good analogy:

> "Science does not rest on solid bedrock. The bold structure of its theories rises, as it were, above a swamp. It is like a building erected on piles. The piles are driven down from above into the swamp, but not down to any natural or 'given' base; and if we stop driving the piles deeper, it is not because we have reached firm ground. We simply stop when we are satisfied that the piles are firm enough to carry the structure, at least for the time being." (Popper, 1990, page 111)

1.5 Guide to the rest of the thesis

"Related work" will not be found in any one place: this research is characterized by the fact that a technique developed for use in one domain had been adapted for use in quite another. Hence we shall meet work related to the domain of configuration, but using different approaches; and we shall find descriptions of the technique I use but applied in different domains.

Chapter 2 gives an account of other automated configurers, for the most part using more conventional expert systems technology, together with a brief account of the history of hardware configuration. Chapter 3 defines the task and describes the theory of configuration as derived from the knowledge elicitation stages of the research, from human experts working in the field and other sources. Chapter 4 is a formal account of the object-level theory of configuration. Chapter 5 provides an account of the proof planning work initiated by Bundy (1987) and describes how inference is guided in the computer configuration task by the use of meta-level techniques. Chapter 6 gives a detailed account of the implementation and testing stages, and Chapter 7 takes up some of the issues and alternatives raised in testing the implementation. Chapter 8 concludes with a summary of what has been achieved, and a look forward to what might be done in the future.

Chapter 2

A historical perspective on configuration

Knowledge cannot start from nothing ... nor yet from observation. The advance of knowledge consists, mainly, in the modification of earlier knowledge.

Karl Popper

2.1 Introduction

Before I expound my particular approach to configuration and the justification for it, it is appropriate to describe both

1. the work of other researchers in this field who use contrasting approaches, and
2. the work done by researchers using an approach similar to my own but in other domains.

I think of the related work as falling into three categories:

1. General work in the field of design, by researchers who see configuration as a subproblem of design.
2. Research into building intelligent knowledge based systems which automate a configuration task. Two tasks which have been tackled are:
   (a) Checking an order for hardware to configure a system.

(b) Generating such an order.

3. Research into theorem proving techniques (such as proof planning) and immediate applications, *e.g.* program synthesis.

The tasks tackled by each system in (2) above are somewhat different, complicating the issue of comparing them with each other and with that which I have designed and implemented. The reason for this is tied up with the changing nature of the configuration task, mirroring the changes in technology which has, over the last two decades,[1] obviated some parts of the task and invented new ones. This has proved to be a fascinating study, showing how technological advances cause economic changes which have profound repercussions on work activities. Parallels can be drawn with Adam Smith's accounts of pin manufacture. In his celebrated account (Smith, 1776) he describes how specialization had completely transformed the task, such that productivity changed from a few hundred to five thousand pins per head per day. Thus our expectations of what can be achieved can be so radically altered by leaps in technological innovation — such as has happened over the last thirty years in the computer industry — that the whole basis of how tasks are performed is transformed. Cheap hardware (and relatively expensive labour) have changed the configuration task from one in which the aim is to fulfill needs stated implicitly as lists of components, as cheaply as possible, into one where we want to be able to explore various design issues. In short, it has changed from a highly constrained problem into an underconstrained one.

In Chapter 5.1 I discuss related work as defined in category (3) above. The discussion of related work as defined by (1) is begun in Section 2.2 of this chapter and resumed in Section 2.4. From there, I shall go on in the remaining sections of this chapter to concentrate on related work (2), that of developing software systems to perform configuration tasks.

---

[1] roughly the period I am considering — from the inception of R1, which was later to become XCON, to the present day

2.2 Definitions of configuration

Over the last decade, the configuration task has evolved into a very different problem than that addressed by the early configuration experts as captured by McDermott (1982). This reflects the extensive changes in computer hardware over the same time period. Both technological and economic factors are involved. In the early days, hardware components, memory, and processing power were comparatively costly and systems such as those of DEC were designed to configure hardware which conformed to minimal (therefore cheapest) customer requirements. The strategies used by experts, which were incorporated into the expert system, constrained the search space in such a way that the answer to "what to do next" was deterministic at every stage.

However, with the advent of cheaper processing power and hardware components, modern computer systems tend to be "over-engineered", in the sense that the basic "frame" of a system has considerable scope for expansion built into it. Thus a configuration built around a particular processor will have ample sockets and extension points to allow expansion of the system to be carried out without necessitating a *field upgrade.*[2] The problem of configuring a computer system to meet requirements has become, in the old sense of configuration, a somewhat under-constrained one.

A major effect of this technological advance has been to render redundant much of the detailed decision-making which originally formed a large part of configuration procedure. These decisions include those which determine the exact positioning of components in sockets (slots or ports). If not completely unnecessary, it can at least be said that this procedure now occupies a minor role as compared with, for instance, determining which of many design principles should be followed. (See Chapter 3.6.3 for a discussion of this.)

---

[2]Changes to the hardware which involve exchanging the existing processor for a more powerful one.

Let us now review various software systems, starting with the earliest such, R1 (McDermott, 1982). I will chart the development of this system and other spinoffs from it, before looking at two very different approaches; namely a constraint based system (Section 2.4), and a logic based system using a semantic net representation (Section 2.5).

2.3 Production rule systems

2.3.1 Basic framework

An expert system based on production rules is composed of the following elements:

1. *Working Memory*: a set of data structures representing the current state of the system.
2. *Production memory*: a set of rules of the form

$$< conditions > \rightarrow < actions >$$

or

$$< working\ memory\ patterns > \rightarrow < working\ memory\ changes >$$

3. A *rule interpreter* which selects which rules to apply by evaluating their conditions.

The *rules* in a production rule system have the form

$$< conditions > \rightarrow < actions >$$

where the left hand side consists of one or more *conditions*, and the right hand side consists of one or more *actions*. At each stage, all rules have their conditions evaluated and those whose conditions evaluate to *true* can, potentially, *fire* — *i.e.* have their actions performed. In general, several rules will have their conditions met. In order to decide which one is actually fired, a *conflict resolution* strategy is used. There are many versions of these: we shall not concern ourselves with the details, or the pros and cons of different strategies, here: the interested reader will find a fuller general account of production rule systems in Jackson (1986).

2.3.2 XCON/XSEL

An account of the original R1 system for checking orders is found in McDermott (1982), while later updates of XCON (as R1 became later known) and XSEL (a companion order generator) are in Bachant & Soloway (1989) and Barker & O'Connor (1989). Soloway *et al.* (1987) give an account of how XCON was reimplemented in RIME (see Section 2.3.3). Rosenbloom *et al.* (1985) gives an account of an experiment using some of the knowledge encapsulated by R1 in the SOAR architecture (see Section 2.3.4).

Expert systems which were designed with older notions of configuration in mind leave task definition implicit within the procedural descriptions of the machinations of the expert system. An example is given in Figure 2–1. This is from the original R1 system, but the various enhancements to XCON suffer from the same problem to an extent.

XCON is a customer order *checker* — it takes an order for a configuration (*i.e.* a list of components) and attempts to configure them, adding any parts which are necessary but which have been omitted from the order, and making other corrections. XSEL works at a higher level, taking account of customer requirements to assemble an order for hardware components, and is interactive, in that the user is prompted for components; the system cannot generate them in any way. At the end of the interactive stage of the process XSEL passes an order to XCON. DEC are hoping to develop a system which starts the process one stage back: *i.e.* at the level of specifying overall goals for the customer, but at the time of writing it is believed that nothing has come out of this yet.

The configuration process as a whole is seen as a sequence of decisions to do with connecting components together and *design* of configurations does not play any role in the automated system, such decisions being left to the user, who must decide which components will be necessary.

XCON, DEC's rule-based production system, and its companion, XSEL, have been successfully used in order checking and configuration respectively. XCON configures VAX-11/780 computer systems, in the sense that it takes a customer order

```
DISTRIBUTE-MB-DEVICES-3

IF:
THE MOST CURRENT ACTIVE CONTEXT IS DISTRIBUTING MASSBUS DEVICES
AND THERE IS A SINGLE PORT DISK DRIVE
    THAT HAS NOT BEEN ASSIGNED TO A MASSBUS
AND THERE ARE NO UNASSIGNED DUAL PORT DISK DRIVES
AND THE NUMBER OF DEVICES THAT EACH MASSBUS SHOULD SUPPORT IS KNOWN
AND THERE IS A MASSBUS
    THAT HAS BEEN ASSIGNED AT LEAST ONE DISK DRIVE
    AND THAT SHOULD SUPPORT ADDITIONAL DISK DRIVES
AND THE TYPE OF CABLE NEEDED TO CONNECT THE DISK DRIVE
    TO THE PREVIOUS DEVICE ON THE MASSBUS IS KNOWN
THEN:
    ASSIGN THE DISK DRIVE TO THE MASSBUS
```

Figure 2–1: An English translation of a sample R1 rule

as input and returns both the modifications to the customer order necessary to obtain a configurable system (if any) and diagrams showing the spatial relationships between the component parts. An account is given in McDermott (1982), Bachant & Soloway (1989), and Barker & O'Connor (1989). In XCON's production rule system, the right hand (action) sides of rules specify how to extend partial configurations, whilst the left hand sides of rules hold constraint information on whether certain partial configurations may be extended. It uses a system of contexts to constrain the applicability of the various rules, and some rules effect a change of context under conditions given by the left hand side. The rules can be divided into three types:

1. Rules to create or extend partial configurations.
2. Rules to determine the order in which decisions should be made.

3. Rules which gather information for other rules, *e.g.* access the components knowledge base or perform computations.

The central problem-solving technique is **Match**, with **Generate-and-test** as a kind of weak backup technique. **Match** recognises what to do next at any point in the program's execution, and the system as a whole never "backtracks" in the sense that decisions are undone; it is sometimes the case, however, that components need to be "unconfigured" if an *impasse* is reached — which *is* actually the same (in effect) as backtracking. Thus it is acting as a decision procedure, and the process is deterministic. Once an acceptable solution is found, the search is finished. This technique works because (it is claimed) the domain knowledge is such that it is possible to determine *locally* whether a given action is appropriate. All the information needed to make a decision is contained in the local context, and decisions yet to be made have effects which propagate only forwards, and not backwards so as to affect decisions already made. Thus very little search is involved, the main task being to dynamically order the set of decisions and generate a single acceptable solution. The knowledge base consists of a large set of constraints and component information. The expert knowledge embodied in the rule set of the system consists of two types of information:

1. The temporal relationships between the subtasks involved: *i.e.* when a subtask should be initiated.

2. Detailed information about when it is appropriate to extend partial configurations in particular ways.

XCON has a reputation for being difficult to maintain. Circumstantial evidence is that in the published accounts (Barker & O'Connor, 1989) the problem of training personnel to work on XCON is prominent; the solution proposed is to send all new staff on an AI course on entry.

The main problem is that one change can often have a large knock-on effect in terms of how many rules need modification. There is no separation of domain knowledge ("facts") and control knowledge ("strategy"). Most of the conditions in the condition-action rules define the context in which the rule is to be considered. Hence control information is implicit rather than explicit, and mixed in with the

factual knowledge. One piece of knowledge may be contained in many rules, so that if this piece of knowledge needs to be modified, then so must all these rules — and they have first to be identified. This is difficult for anyone but an experienced maintenance programmer — experienced, that is, in XCON itself. "As the size of the knowledge base increases, it becomes even more difficult to determine all of the interrelationships between the rules" (Freeman, 1985). By 1987 XCON had 6,200 rules, a tall order for maintenance. However, it is seen as a successful system, Harmon (1987) citing that 99% of its 50,000 orders in 1986 were properly specified. Accounts, generally positive in tone, of the course of development of XCON and XSEL, are given in Mumford & MacDonald (1989) and, for a shorter account, Polit (1985) These accounts, the first in particular, give the systems analyst viewpoint of how new technology can be successfully introduced. The secrets of success according to Polit are, firstly, not making inflated claims in advance of what can be achieved and, secondly, attracting a critical mass of AI expertise in the company.

2.3.3 XCON in RIME

Attempts have been made to improve the control aspect of XCON, as described in Soloway *et al.* (1987), by using a higher level language, namely RIME, where the relationship of RIME to OPS5 (in which XCON is written) is analogous to that between FORTRAN and assembly code. The main claim made for XCON-*in*-RIME is that control knowledge is made explicit: both in terms of the problem-solving method (claimed to be domain-independent); and the entering of different problem spaces or *contexts* (said to be domain-specific).

However, even with such improvements, it still remains the case that use of domain knowledge could not be extended, say to automating the process from user specifications to configurations, or to teaching configuration, even though the same domain knowledge would be used for all three tasks. The main problem is that the central strategy, which is fundamental to the XCON system, is specific to this particular method of configuration checking, for these particular systems. It depends for its success on the fact that any local solution to a problem is always correct

in the global context. Thus if, for instance, we find a solution to the subgoal of configuring the backplane, say, then there will be no conflict with any other subgoal later on. Also, the subgoals are each tightly constrained, so that although there may be several solutions (note: not hundreds, except with regard to trivial permutations) each one of them is equally good. This approach cannot be used in more open-ended problems where there are many solutions, but only a few *locally optimal* solutions. [3] In my configuration system, it proved necessary to consider various classes of solution; for example:

- the *best-performance* solutions for a fixed cost;
- the *cheapest* solution meeting a given specification.

These questions necessitate the principled inclusion of heuristics which can direct search efficiently towards these local optima. Without this, then because it is not the case that a solution to a local subgoal will guarantee a complete, global solution, much search will be involved.

Thus although overall control can be made much more explicit using RIME, it still cannot properly separate the use and control of heuristics, of increasing importance in modern systems.

Moreover, constant updating of XCON and XSEL is a feature of rapidly changing product lines, and, as described in Barker & O'Connor (1989), specialized training of personnel is essential to carry out this task. This reflects the fact that there is insufficient separation of control knowledge from factual and heuristic knowledge. The task of maintaining the system is not straightforward, needing specialized knowledge of the computer system as a whole, and an understanding of AI techniques. Personnel responsible for product updating cannot perform their task in isolation; likewise neither can configuration engineers responsible for maintaining performance heuristics and control strategies.

---

[3] I define this term formally in Chapter 3.6.4. The definitions may also be found in Appendix A.

I do not believe that an intricate knowledge of expert systems should be needed for routine maintenance, and have directed my efforts to designing a system which can carry out the same tasks as XSEL but with more possibilities for expansion (notably by allowing higher-level goals to be input, once the knowledge to support the necessary inference becomes available), and which will not have the same maintenance overhead as is inherent in production rule systems.

2.3.4 R1-SOAR

First let me stress that, to the best of my knowledge, R1-SOAR has not been used in a commercial setting, and was never intended to be. Rather, the domain of R1 was used, as an example of a large knowledge based application, to test whether SOAR could be used for such an application. Nevertheless, it is useful to examine the claims made by Rosenbloom *et al.* (1985) that such an architecture could prove to be a practical proposition for such a domain, especially as, at a superficial level, it bears some resemblance to the architecture of a proof-planning system.

Rosenbloom *et al.* differentiate between domain-dependent *knowledge intensive* and domain-independent *general problem-solving* approaches. The pure R1 system is an example of the former, while SOAR's basic architecture provides a more general problem-solving framework. The idea is to add to this basic framework enough expertise, in the form of rules which control search, to enable SOAR to carry out the same task as R1, but more efficiently. Various amounts of expertise can be added to SOAR, from none (so that it is acting in pure general problem-solving mode) to a lot (so that it is knowledge intensive), with the general problem-solving framework always present at least as a backup.

R1-SOAR can carry out about one third of the task of R1, using 1100 of its 3300 rules. Experiments were carried out on four tasks, using various degrees of expertise, from using only SOAR's general problem-solving methods to a version using much of R1's domain knowledge. All but the simplest examples required *some* expertise to be properly workable.

R1-SOAR partitions the task into problem spaces, in a hierarchical manner — for example, the top space has the goal of configuring the entire system, while subspaces deal with subcomponents of this task, such as configuring a backplane. Apart from this, there seem to be the same problems as with R1 associated with adding rules to a production rule system. There is nothing to suggest that this framework does not allow intermingling of facts and control knowledge in the conditions of rules. Since this system has not been used commercially, it remains to be seen whether maintenance programmers would be encouraged by this system to exercise discipline in the implementation of these different types of knowledge.

Also, this hierarchy of problem spaces is inflexible when it comes to generalizing the problem — for example, during system upgrading knowledge about the peripherals may be utilized before details of the major system structure is known. My *methods* (Chapter 5) avoid this built-in inflexibility.

It is not clear that this system can be readily maintained, nor even that it will scale up to perform the whole of the task which R1 carries out. There seems nothing to prevent rules being added in an *ad hoc* way, and an understanding of all the rules and their usage and implications would seem to be necessary for maintenance of all kinds. Moreover, SOAR's ability to acquire new knowledge by a process known as *chunking* (see Laird *et al.* (1987) for a full account) is a double-edged sword, leading to problems of overgeneralization. An example of this given by the authors is where the system surmises, from the fact that a module cannot be configured on a particular backplane, that it cannot be configured on *any* backplane. This seems a serious drawback.

2.4 Constraint-driven systems: COSSACK

I turn, therefore, to a more recent view of the configuration task. Here the expert systems developed are more prototypical than XCON, but the task definitions are more explicit. The authors of COSSACK (Frayman & Mittal, 1987) define the configuration task thus:

> "Given a fixed, pre-determined set of components, an architecture that defines certain ways of connecting these components, and requirements imposed by a user for specific cases, either select a set of components that satisfies all relevant requirements or detect inconsistencies in the requirements."

Configuration is here thought of as a special case of design. In fact, it falls within the category given in Chandrasekaran *et al.* (1987) as "object synthesis by plan selection and refinement", whose task specification is to

> "Design an object satisfying specifications (object in an abstract sense: they can be plans, programs, *etc.*)"

The restriction over design activity in general is that the components are pre-defined and can only be connected together in certain ways. A fuller definition of the generic configuration task is described in Mittal & Frayman (1989):

> "**Given:**
>
> 1. a fixed, predetermined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints
> 2. some description of the desired configuration
> 3. possibly some criteria for making optimal selections
>
> 1. Build one or more configurations satisfying all the requirements, where a configuration is a set of components and a description of the connections between the components in the set
>
>   or
>
> 2. Detect inconsistencies in the requirements."

The term "port" is used here as an abstraction for a place where components may be connected. Note that a configuration is not simply a set of components, but must also include (somehow) information about how these components are to be joined together.

The authors claim that this definition is generic in that other tasks fit into it, for example the design of single-board computer systems and even the choice of a car.

I shall not consider here whether the wide generic view is appropriate, or indeed useful, but will return to this question in Chapter 8. Section 8.4.

Frayman & Mittal have developed a knowledge based system known as COSSACK for configuring XEROX PCs. The configuration task is restricted in that artifacts are configured according to known functional architectures: those permissible for a Xerox PC are explicitly represented and COSSACK instantiates one of these using a set of standard components, building up the configuration around *key components* — the devices of the configuration — which act as "planning islands" of *almost* independent subsystems. There is still some interdependence, however, in that devices share components, and each device may perform more than one function.

The basic strategy is to start with the functions required (*e.g.* printing, to a certain specification) and map these on to key components (*e.g.* a specific printer). This component "posts" requirements which either call for further functions or else places constraints on existing ones. The second stage is to seek a consistent connection of the components. These stages can be interleaved to prevent expensive backtracking. Experiments have been tried: using heuristics, and the use of *partial solutions* — *i.e.* sets of components which fulfill a function rather than picking one.

COSSACK is a system strongly reminiscent of MOLGEN (Stefik, 1981) which essentially treats configuration as a constraint-driven problem. As such, it is eminently suitable for systems which are tailored very specifically to the specification, but not to systems which are in some sense "over-engineered": these modern architectures aim to provide scope for expansion and hence tend to be highly under-constrained. The examples given in the paper cited above are of relatively small computer systems only. Since no examples of larger hardware configurations are given, it would appear that the search space associated with a system in the style of COSSACK would be too great, and the number of solutions generated too large, for tackling problems of such an order of magnitude.

2.5 Logic based approaches: BEACON

BEACON (Searls & Norton, 1990), based on a semantic network, and implemented in Prolog, configures Unisys computer systems. This implementation was aimed at utilizing a more declarative approach to the configuration task, in contrast with the systems we have seen so far. The authors concern themselves with maintainability; by then it was becoming apparent that this was the weakness of XCON, then still apparently seen as the torchbearer for automated configurers. I explained the advantages of a declarative approach in Chapter 1, Section 1.2.2, and this is seen as the key here too.

Searls & Norton recognize the need for what they call *functional* configuration — configuring to specifications couched in terms of higher level goals than merely what devices are needed, for example the requirement that there should be enough disk space to support the administration needs of a typical 200-bed hospital. They also call for the task of the sales representative and the task of the engineer to be integrated, rather than separate as is (still) common practice. The latter was the main contribution of BEACON, which is an interactive configurer where the constraints are dealt with automatically and the user presented only with feasible (legal) options. It is not clear, however, how the functional approach at the high level suggested should be implemented.

BEACON uses the semantic network KNET and an interactive ordering system. The user is presented with choices, all of which are guaranteed correct by the underlying configuration rules used in drawing up these choices. The inference strategy used is in the spirit of Prolog-style search. As the process continues, a configuration is built in a manner analogous to the instantiation of variables in a Prolog-style proof.

The approach is a useful one, in that it is logic-based, and the constraints are integrated into the logic so that, firstly, the user is never presented with illegal choices, and secondly that there is no expensive Prolog-style backtracking — which

would be the case if the constraints were checked after the components had been chosen.

However, the examples given for the performance of this system are of configuring microcomputers and mini-computers which come in a limited number of sizes. It does not seem likely that this system will scale up without some kind of meta-level guidance.

2.6 Hybrid systems

Pierick (1986) believes that neither a production rule system nor a semantic net system can adequately represent such a complex domain as configuration, and proposes a a hybrid system, combining the two. However, he is reluctant to discard production rule systems altogether, as most commercial expert systems rely on them.[4] However, he points out the disadvantages of lack of modularity and the increasing complexity of the interrelationships between rules, as explained in Section 2.3.2. He proposes a frame-based system (Brachman, 1983), with a frame to represent each distinct component. A complete system is also represented by a frame, with a slot containing the names of the frames representing its components. More generally, any component's relationships with other components is represented by a slot in its frame. Information about any given component is found, and modified, uniquely in the frame representing that component. Use is made of the inheritance facility of frames to allow knowledge to be located at "the most logical" place. For example, the fact that an IBM PC has a display is not held in the frame for the specific model, but in the frame for PCs in general, since *all* PCs have displays.

However, Pierick finds that, while frames are good for representing the "structure" of configuration, they are not good for representing configuration "procedure"

---

[4]Perhaps this, as much as anything, demonstrates the power of an existing paradigm on development (see Chapter 8, Section 8.5).

and hence are only of limited use, perhaps only to experts who are intimately familiar with the task (and, therefore, have no need for an automated system). In particular, a user needs to know when, exactly, to apply various constraints. As we shall see in Chapter 3, Section 3.5, many of the constraints in configuration are not "hard" but "soft" and the skilled engineer will know when, and when not, to apply them. Pierick calls this "judgemental reasoning". For example, the user can trigger "if-needed" production rules in certain slots to decide whether a particular component is to be used. This production rule dæmon solves a highly focussed problem, using, where appropriate, certainty factors to guide the user towards appropriate hardware dependent on their needs.

As with other systems described in this Chapter, the examples given are of small systems. The use of certainty factors in particular seems arbitrary: Pierick talks of "empirical" knowledge but it is not clear how the values of the certainty factors would be determined, even empirically. The rapid turnover of components in the domain of computer hardware would seem to exacerbate the problem beyond even the difficulty faced by, say, the medical domain — knowledge does not grow, but simply change.

We shall see in (Chapter 4, Sections 4.4.2–4.4.5) that the configuration domain poses problems of multiple inheritance — there may be more than one way of classifying a component. Pierick's paper does not appear to recognize this problem.

It could be said that such a system combines the worst of both worlds, rather than the benefits of each.

Another exponent of multi-representational systems is the paper of Wu *et al.* (1986). Their Intelligent System Configuration Shell (ISCS) uses several different tools and knowledge representations glued together by the use of an object-orientated programming environment. It has separate modules: one providing a "knowledge engineering" assistant; another for control and inference; others for maintaining the database and providing a user interface. This system recognizes the maintenance problem and this shell is used for this purpose, as well as for the actual configuration task. It uses a taxonomy for classifying components and allowing inheritance of certain attributes and values in a manner similar to that

described by Pierick. It also uses dæmons for constraints which may be introduced by the user, and to allow the system to select an appropriate action if constraints are violated. For example, if there are not enough lines available for all the devices, another will be added. Constraint propagation makes sure that this addition does not violate any further constraints, and so on. Amongst the procedural knowledge possessed by the system is a "plan" for a task, which decides which rules are to be executed, in which order. This is hierarchical, in that plans may themselves be modified, by "meta-plans".

2.7 Summary

XCON and XSEL have served their purpose well and represent good examples of traditional production rule expert systems. However, the problem of maintenance is a vexed one. Production-rule systems seem to be well entrenched at the commercial end of development; nevertheless others have attempted to use systems which either incorporated production rules alongside alternative representations, or broke with the production rule paradigm altogether. Systems such as BEACON recognized the need for systems of a more declarative nature. The use of a Prolog implementation, at least at the prototype stage, is useful for these purposes. Although BEACON is a step in the right direction in placing configuration on a formal logical basis to guarantee the soundness of the resulting configurations, the question of meta-level knowledge to control the search problem is not addressed. However, this question is crucial for large multi-user systems. SOAR seems rather unwieldy and does not appear to solve either the soundness or the maintenance problem. COSSACK seems unlikely to be able to cope with some of the search spaces which are common for large hardware systems. Obviously the question of how far design issues can be defined and represented is an open one, but assuming that this could be done at some level (I have gone some way towards it using *supermethods*, as explained in Chapter 5), these systems do not seem amenable to embracing such issues.

More recent approaches to building knowledge based systems and to confiugration in particular have stressed the knowledge elicitation alongside the maintenance problem, and the need for careful methodology and organization. One approach which has proved increasingly popular is that of KADS (Wielinga *et al.*, 1992). KADS models consist of four hierarchically ordered layers:

1. Domain layer: knowledge about the application domain, represented independently of *how* it will be used.

2. Inference layer: how to use the knowledge from the domain layer, held indepenenly of *when* it will be used.

3. Task layer: to specify control over the use of the inference steps represented at the inference layer.

4. Strategy layer: how to choose between two tasks which achieve the same goal. This is the least developed aspect of KADS currently.

To a degree, my approach echoes this methodology in that it too stresses the importance of the declarative representation and separation of knowledge. I represent facts about configuration (domain layer), rules for configuring system (inference), preconditions for applying these rules (task layer), and proof plans for putting these together (strategies).

The architecture I propose in this thesis will cope with the large search space associated with such system configuration. The use of a formal object-level representation ensures the legality of the computer systems synthesized. It uses separate control mechanisms to maximize the chances of finding a path through the search space to solutions which are preferable in some way. Strategies for design can be absorbed on an incremental basis.

The ideal is a system which could combine logical soundness with a modern approach to the task, and still, potentially, prove as useful as XCON. Given the varying interpretations, I need to describe precisely what is involved in modern-day configuration. Such a description is found in Chapter 3.

Chapter 3

Configuring computer hardware systems

Theories are nets cast to catch what we call 'the world': to rationalize, to explain, and to master it. We endeavour to make the mesh finer and finer.

Karl Popper

3.1 Introduction

Many knowledge based systems suffer from the difficulty of separating object-level and meta-level knowledge. Knowledge bases are compiled with the aid of human experts who are often unable to unravel the two, or even understand the distinction. As new objects are added or new rules discovered it becomes increasingly difficult to maintain the system. Moreover, new applications, which ought to be based essentially on the same factual knowledge as the old one, cannot use the existing knowledge base, since the information therein is irretrievably bound up with high level control knowledge used by the original application.

I have considered the tasks associated with computer systems configuration. Although I have focussed on configuration synthesis, (*i.e.* configuring computer systems to meet given specifications) I have been aware that there are related tasks which use the same object-level knowledge — namely:

- customer order checking: given an order, ensuring that a viable computer system can be built from it, and

- system upgrading: given an existing system, and new specifications, building a new system based on the old one.

I have endeavoured to separate out and formalize this object-level knowledge so that it may be used by planning systems to either of these ends, or for other tasks in this domain. As far as I know, little or no work has been done on the automation of the upgrading of hardware systems, despite the existence of knowledge based systems for order checking and, to a more limited extent, configuration synthesis. Having achieved this separation, I have gone on to look at the meta-level, task-specific knowledge needed for configuration synthesis.

As intimated above, the task I shall be concentrating on for the most part is that of configuring computer hardware[1] to meet a given specification. It is necessary first to define precisely what is involved in carrying out this task for modern hardware systems since, as explained in Chapter 2, configuration strategies have changed in line with the leaps in technology of the hardware itself. I need also to place the specifications in the context of the negotiations involved in drawing them up (see Section 3.2.5). This chapter will illustrate typical specifications and possible candidate configurations which I would wish to be able to generate, and will describe the informal reasoning processes of experts which give rise to such configurations.

---

[1]Hewlett Packard hardware systems are used to exemplify the substance of this and subsequent chapters. I am extremely grateful to Hewlett Packard for allowing me access to this information. Please note that most examples are hypothetical, although realistic in that they are generally based on actual cases, and that opinions expressed are my own, and not those of Hewlett Packard.

3.2 The process of hardware specification

3.2.1 Informal assessment of needs

Customers for computer hardware generally think in terms of the use to which the system is to be put, and they will go through the process of discussing customer needs with the sales representative for the computer firm. Issues include the applications to be run, the number of users involved, and possibly other considerations such as cost, and the future needs of the company as regards expanding or upgrading any hardware configured.

Sometimes there will be a formal *invitation to tender*. Here, the customer presents the requirements in written form and computer firms are invited to submit tenders, detailing the hardware to be configured, itemized costs, and indicating parts of the configuration which either fall below or exceed the specification, with reasons.

A typical (but fictitious) specification drawn up at the invitation to tender stage is shown in Figure 3–1. Using this as an example, note the following issues.

- 70 screens for data entry
- 5 screens for program development
- 20 screens for electronic mail and spreadsheet application
- 1 line printer with a speed of at least 900 lpm
- 2 laser printers, each with a speed of at least 45 ppm
- 1 laser printer with double-sided printing option
- approximately 1.7 GBytes of mass storage

Figure 3–1: Invitation to tender specifications

3.2.2 Choice of processor

The customer may or may not specify the processor to be used. Of course, in the case of general invitations to tender the customer *cannot* specify a processor since the invitation to tender specifications are general and aimed at competing hardware firms. However, it is usual for the representative of the tendering firm to decide on the processor first. To do this requires certain domain knowledge and the ability to assimilate salient facts from the specification as a whole. For instance, the total number of ports required in the above specification is 95–99 (95 for the screens; possibly up to 4 ports for the printers if any or all of them are configured as serial devices) and rules out all but the largest computer systems. We would wish any automated system to be able to cope whether or not the processor is specified.

3.2.3 Devices

Disk storage devices

Disk storage requirements may be specified at various levels.

1. The total disk storage capacity required is given, leaving the exact configuration of drives unspecified.

2. Disk drives could be explicitly specified, *e.g.* ten model *example_ddrive* drives. This level of specification is not typical since it is more detailed than is usually possible.

3. The total capacity is given, as in 1, together with some other attributes:

   (a) The type of disk drive, for example that *fibre-optic*[2] disk drives are required.

---

[2]See Glossary of Hardware Terms

(b) The access speed required: this may be a qualitative measure rather than an explicit figure.

(c) The minimum (or maximum) number of disk drives to be configured, *e.g.* we require at least two disk drives.

4. It is possible that, in specifying the disk storage in any of the ways above, we might want to specify *user* disk storage separately from *system* disk drives.[3]

Terminals

The terminals of a configuration could be explicitly specified: *i.e.* that we require n_1 of terminal model *term_1* and n_2 of terminal model *term_3*. However, it is more usual to specify terminals in terms of applications, or simply to state the total number of terminals required for the computer system.

Printers

The same remarks apply for printers as for terminals; in addition, it is possible, as in the example of Figure 3–1, to specify each printer required by its attributes. For printers, these attributes can be: *speed* (lines per minute; characters per second); a measure of the robustness of the printer in terms with the volume it can cope with, known as *throughput* (pages per month); availability of certain features like the ability to offer double-sided printing, high quality printing, *etc.*

Tape drives

Tape drives in a computer system have two main uses: backup, and transfer (of data, software, *etc.*). If backup is the only consideration it is unusual for the tape drives to appear in the specification. Instead, the tendering firm is expected to

---

[3]See Glossary of Hardware Terms for an explanation of the terms *system* and *user* disks.

assess the backup needs of the configured system and suggest suitable tape drives. The main consideration then is the disk storage capacity of the system to be backed up. Some of the disk storage will be used for data which is never backed up: for instance, swap space. The total disk storage capacity therefore gives a convenient upper bound for the amount of backup capacity required; the suitability of various tape drives for various storage capacities is a known attribute of the drives.

3.2.4 Memory

It is unusual for the amount of memory required to be specified. In particular, the amount needed depends partially on the operating system used, which in turn is dependent on the processor used, amongst other factors. So a customer inviting tenders is unlikely to be able to specify memory and, as for tape drives, will expect the tendering firm to work out how much memory is needed.

For HP systems, each processor has a formula

$$memory\_capacity = F + M \times N$$

where F is a fixed amount depending on the operating system used by the processor, N is the number of concurrent active users (or the maximum of this, if it is variable), and M is a number in a range $[\alpha, \beta]$, whose position in this range is a function of the applications — some applications requiring more memory than others. Actually, "function" is probably an over-precise term for what is, essentially, rule of thumb. Since memory boards come in discrete sizes (1Mb, 4Mb, *etc.*), as long as we end up in the right band further accuracy is unimportant.

3.2.5 Tendering considerations

In the tendering situation, each computer firm is in competition with others. However, this is not the straightforward situation where the lowest bid automatically wins. The specification is, to an extent, open-ended and it is up to the tendering firm to state what, exactly, is being provided for the money. The aim

in tendering of this nature is to provide the best value for money, compared with the competition.

There are several points of interest here:

1. There are a very large number of possible configurations which will meet the specification given.

2. They are not all of equal merit.

3. There are a number of dimensions on which to judge solutions: for instance cost, performance criteria, expandability.

4. Because of this there is not, in general, a *global optimum* solution (see Section 3.6.4).

5. We would hope the system would discard any solutions which were "locally sub-optimal" (see Section 3.6.4); *e.g.* we could find another solution which gave better performance, cost and all other criteria being equal.

We want to be free to reason about how to arrive at locally optimal solutions safe in the knowledge that the underlying object theory of configuration is sound and complete.

All these issues will be addressed in the succeeding sections.

3.3 Description of 'legal' systems

3.3.1 Generic computer systems configuration

A formal definition of a *legal configuration* can be found in Chapter 4, Section 4.7. What follows is a relatively informal discussion of the issues involved in drawing up such a definition.

Although I have no pretensions of directly addressing the generic task intimated in Chapter 2, Section 2.2, where configuration problems were seen as a subset of design problems, at the very least I wish to be able to describe the task in such a way that it addresses a large class of modern hardware systems. Even when confining oneself to a single manufacturer these systems can appear disparate — from small, single user systems to large mainframes. Between manufacturers, vocabulary can differ to the extent that it becomes difficult to believe that we have the same domain. Our task is to develop a language which draws out the commonalities between these different systems — a standard language for computer hardware configuration.

We must accept that the procedures for configuration may differ between systems, and particularly between different manufacturers. However, at a high level, I believe that the task is the same, and I wish to take this commonality as far as possible, so that individual procedures may be described in terms of their effects and the functionality of the computer hardware system ("what it does").

3.3.2 Commonality between computer hardware configurations

All computer configurations will have values for their attributes: memory capacity; storage capacity; printers configured; number of user ports; cost; and so on. However, to draw up a general description of what constitutes a *legal* configuration needs careful thought. Consider the incomplete tree of components as shown in Figure 3–2. Appendix B.1 (Glossary of Hardware Terms), gives a list of some of the components we shall come across. We identify the component types "processor" and "memory", which are generic in the sense that all computer hardware systems possess some kind of processor and some kinds of memory boards. All computer systems will contain "devices" also. However, immediately we consider the subdivision of this last category we are in difficulty. Some systems will demark disk and tape storage: others will not, having devices which combine the functions of both these kinds of device. Printers and plotters are untroublesome,

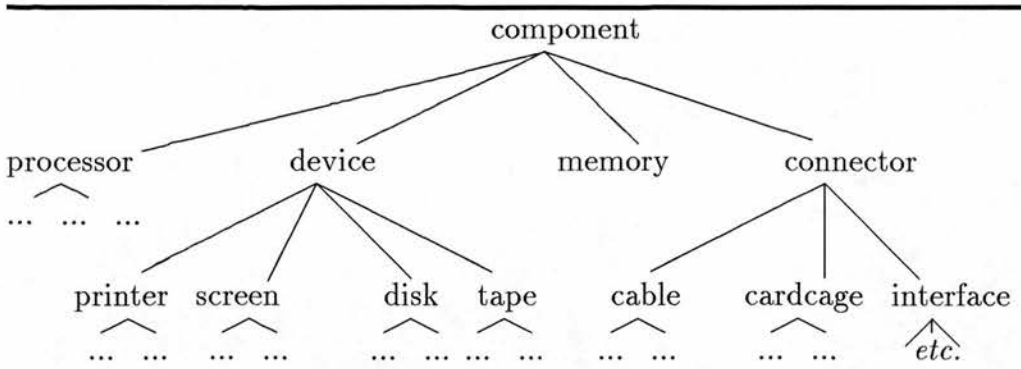

Figure 3–2: Component tree

but "screens" are not. All users must have some means of communicating with the system, but whereas some will use terminals others may use personal computers (PCs) — and PCs possess processing power, maybe disk drives, and so forth. We can make a start at drawing a tree of component hierarchies, but the waters get fairly muddied when it comes to producing generic component trees.

Because of this, I propose a less detailed, simpler generic component tree which is shown in Figure 3–3.

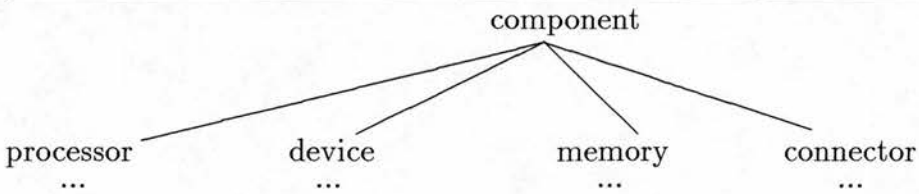

Figure 3–3: Generic component tree

In order to produce this tree, I identified the following rules which all computer configurations must obey:

1. All devices present must be connected.
2. Each device has a *unique* connection.
3. All interface channels must be connected.
4. There must be at least one processor.

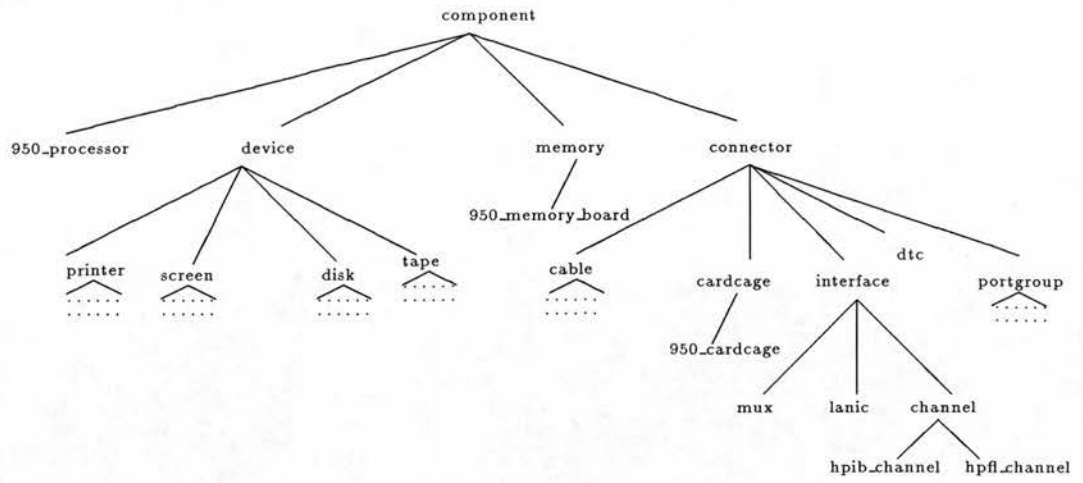

Figure 3–4: Part of the component tree for the HP 950 processor

5. There must be a console for communicating with the system.
6. There must be some memory.
7. There must be a means of storing the operating system (and other software).

Some points of clarification should be made here. In (1), we need to define the concept of *being connected.* For (2), it should be noted that a device cannot be simultaneously connected via two different means. Thus for a given device, it will have one channel of communication (a cable, modem, or some other component of this nature), and this will be connected *via* a suitable interface exactly once. Naturally, it is possible to connect a device by alternative means; for example, we could unplug a printer and connect it differently; this represents a different configuration, however. In (3) there is, again, one, and only one, connection: each interface resides in exactly one slot. In (5), "console" is an abstraction for whatever device is appropriate or permissible: it is processor dependent, in general.

In summary, these requirements are all couched in very general terms, and deal mostly in the *functionality* of the configuration rather than in terms of components.

I make the observation here that the choice of processor has the effect of restricting or even fixing many choices — which types and models of devices are permissible

(*i.e.* that the processor *supports*); what means are available for connecting them, and so on. The difficulties with drawing up component trees largely disappear, and part of one for the HP 950 processor is shown in Figure 3–4.

In fact, more than this, choosing a processor effectively sets up a kind of "template configuration": the HP 950 template is shown in Figure 3–5, and a contrasting template for the HP Series 70 is shown in Figure 3–6. The expert configurer, confronted with a specification such as the example given in Figure 3–1, apparently makes a leap from the user requirements to a processor (or a candidate processor). For example, for the requirements given in Figure 3–1, the number of users, combined with a high disk storage capacity, point, for the experienced sales representative, to a 950 processor. On the other hand, the processor may be specified from some other criterion; for example, if we know that the configuration will be upgraded in the future we might decide to specify a processor more powerful than that suggested by the other user requirements explicit at the time. Thus we see that the processor may be either explicitly specified, or else inferred from other relevant information.

In summary, there is a small class of template systems. The choice of processor determines the template, the devices which may be supported, and the connecting components which are available; and imposes various constraints, such as the number of extra card cages which may be added, and the number of interface cards which may be configured in each card cage.

3.4 Object-level rules

3.4.1 Device models and attributes

In order to determine, say, the total disk storage capacity of a configuration, we record certain attributes of the devices involved: for example, the capacity of a disk drive in megabytes. These properties belong properly with the model of device,

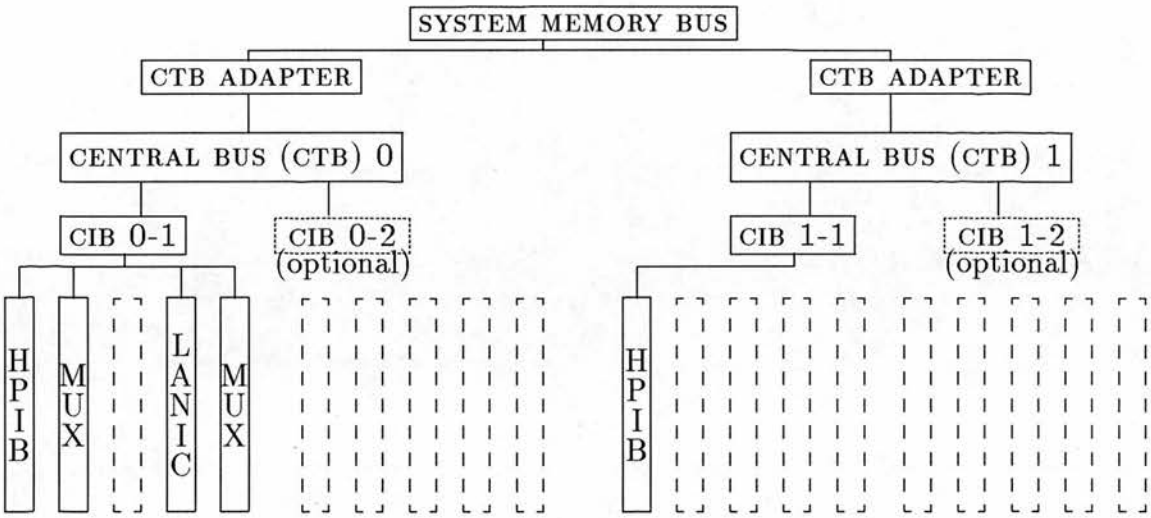

Figure 3–5: HP 950 template

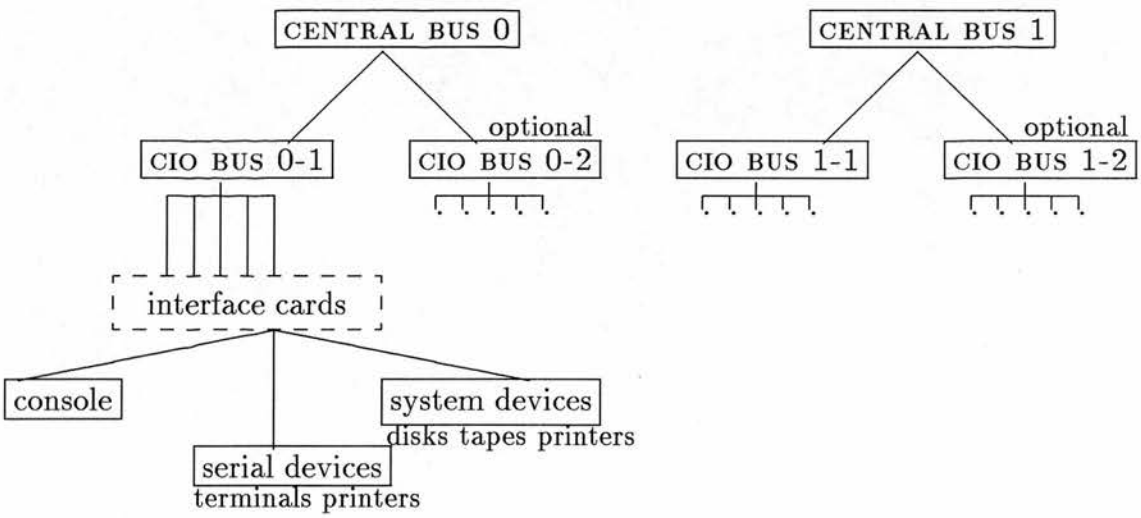

Figure 3–6: HP Series 70 template

to be inherited by individual devices in the configuration. Properties which are recorded in this way are

1. Capacity of
   (a) memory boards
   (b) disk drives
   (c) tape drives
2. Speeds of printers in
   (a) lines per minute
   (b) characters per second
   (c) pages per minute
3. Throughput of printers in

   pages of output per month
4. Mode of operation of printers
   (a) line printers
   (b) page printers
   (c) laser printers
5. Capabilities of terminals
   (a) text/graphics
   (b) suitability for various applications

3.4.2 From generalities to specifics

In Section 3.3.2, I drew up a set of rules which *all* configurations must follow, in order to be counted as *legal.* For a given configuration, a general rule, such as that a device must be connected, is translated into more particular rules: that it must have a cable of a compatible type, and that the other end of this cable must be connected to a port or a channel, again of a suitable type. There are constraints on which cables may be used with which devices, and on the connection method generally. For example, in some configurations a terminal will always be connected via a cable to a MUX; the MUX interface resides in a card cage slot. In others, it is

more complex: a terminal must be connected via its cable or a modem to a port; ports are configured in groups of six or eight on DTCs; and DTCs are configured on a LANIC, whose interface occupies a card cage slot.

In Section 3.3.2, I intimated that each processor was to be associated with a template, shown diagrammatically for two examples (Figures 3–5 and 3–6). This sets various constraints in the form of limits on the numbers of components of one type which may be associated with components of another type. I shall call these *limit constraints.* Here are a few examples:

1. A 950 processor must have at least two card cages.
2. A 950 processor may not have more than four card cages.
3. A card cage in a 950 system has slots for up to five interface cards.
4. An HP-IB channel may not have more than four disk drives connected.
5. An HP-IB channel may have up to six devices (disk drives, tape drives, and printers) connected in total.

There are also constraints to express that certain devices are incompatible: for example, in 950 configurations it is stated that the system disk drive (used to store the operating system) may not share a channel with the system backup tape; in Series 70 systems certain devices are designated as high speed devices and may not be configured alongside low speed devices. I shall refer to these henceforth as *compatibility constraints.*

However, note at this point that these rules are not sufficient in themselves to ensure that the configuration will satisfy all the needs of the customer/user. They ensure merely that the configuration can be assembled and that certain minimal requirements are met, such as there being a console to enable communication with the system, storage for the operating system, and a tape to enable software to be loaded and back up data. They do not ensure that the users will be able to run their software efficiently, or indeed at all. The related question, of achieving "better" systems, or just "adequate" systems will be addressed in Sections 3.5–3.8.

3.5 Meta-level heuristics

3.5.1 Strengthening constraints

Some heuristics are, in effect, a tightening or strengthening of existing rules. They all use heuristic knowledge about what will guarantee an *efficiently running configuration*, as opposed to a simply *legal* one. For example, I said in Section 3.4.2:

1. A card cage in a 950 system has slots for up to *five* interface cards.
2. An HP-IB channel may not have more than *four* disk drives connected.
3. An HP-IB channel may have up to *six* devices (disk drives, tape drives, and printers) connected in total.

These can be thought of as "hard constraints" — the limits stated may not be exceeded in any circumstances (it is probably physically impossible to exceed them; for example, if there are only five places to insert cards in a card cage). However, we have these corresponding 'soft' constraints:

1. The *average* number of interface cards per card cage in a configuration should not exceed *three*.
2. An HP-IB channel may not have more than *three* disk drives connected.
3. An HP-IB channel may have up to *five* devices (disk drives, tape drives, and printers) connected in total.

We can think of *hard constraints* as being the rules which *all* configurations *must* obey, and of *soft constraints* as being rules which *most* configurations *should* obey, *if possible*.

Note that Number 1 above looks (and is) strange and will be explained in Section 3.7 — it is an example of "compiled knowledge": compiled in the brain of the expert configurer, that is.

3.5.2 Acquired knowledge

Sometimes soft compatibility constraints are discovered during the lifetime of a component. For instance, there is a certain model of printer which, when connected to the same channel as a disk drive, causes the computer system to run slowly if both are operational at once. The soft constraint states that this model of printer should not be connected to any channel which has disk drives connected to it.

This type of heuristic information tends to be accumulated over the lifetime of a system product. For example, in the example above, the knowledge that this particular printer caused problems might only be known after feedback from engineers and customers. A new model of printer might come on the market whose performance in certain situations had not been predicted. We can lose these heuristic constraints too — for example, as a result of feedback from customers and engineers, a component may be altered to obviate the problem. It is almost as important that such constraints can be removed when they are no longer appropriate as it is that they are incorporated in the first place.

3.6 Search issues

The problem of configuring a modern computer hardware system to meet user requirements is, in general, sufficiently under-constrained to be problematical. Specifications are often expressed in terms of the functionality of the configuration, and there may be many alternative sets of devices, all of which give the desired functionality. Moreover, there may be many ways of connecting a given set of devices into the configuration, Some of these alternatives will be trivial permutations of others; other alternatives may affect the performance of the system, or its cost.

I can try to constrain the search in these ways:

1. Trying to apply all the soft constraints, if possible.

2. Suppressing trivial alternative solutions.

3. Attempting to return well-designed computer configurations.

4. Attempting not to return locally sub-optimal solutions.

I address each of these in turn in the sections which follow.

3.6.1 Avoiding trivial permutations

If a solution is found, we do not wish to consider, as alternatives, solutions which are in some sense permutations of the first. An example will illustrate this point.

Suppose we have a simple configuration, consisting of two card cages, two channels for configuring disk drives, tape drives, and printers, and two such devices: a disk drive and a tape drive. Since the disk drive is to be used to hold the operating system and the tape drive will provide back up facilities, we have a rule which states that we cannot configure both these devices on the same channel. A solution is found which configures the disk drive on channel 1 and the tape drive on channel 2. We are not then interested in the permutation which configures the devices the other way round.

3.6.2 Applying soft constraints

The principle of applying soft constraints is good in cases where it is possible to find solutions which obey all the soft constraints. If we can find such a solution, where all the user requirements have been met and, in addition, we have been able to apply heuristics which ensure that the configuration will run efficiently under every conceivable circumstance (this heuristic is the principle "apply soft constraints everywhere"), then we have no wish to return solutions which do not meet this standard. In these circumstances, we would wish to prune the search tree so that only the superior solutions are found.

However, it is sometimes the case that if we carried out such a pruning, we would prune all branches leading to solutions from the tree. In this case, we have to

be able to dynamically relax one or more of the soft constraints, on one or more occasions. This is dealt with in Section 3.7.

Returning to the example in Subsection 3.6.1, suppose that in addition to the hardware mentioned there we have a printer of the type mentioned in Section 3.5.2: this was the one which could sometimes cause problems if connected to the same channel as a disk drive. We prefer, if possible, that this printer should not share a channel with the disk drive, but since this is a soft constraint we are prepared to relax it if necessary. In this case, it is not necessary: we can configure the disk drive on channel 1 and the tape drive with the printer on channel 2. We would wish to suppress the trivial permutation solution of configuring the disk drive on channel 2 and the other two devices on channel 1.

Note also that we would *also* wish to suppress the solution of configuring the disk drive and the printer on (say) channel 1 and the tape drive on the other channel, since it is an inferior solution and we have already found a better one. The issue here, however, is achieving a local optimum, and this is dealt with in Section 3.6.4 below.

3.6.3 Achieving good designs

Issues of what constitutes good design are subjective up to a point. I will assume only the following principle in incorporating design decisions:

- There are various "rules of thumb" — heuristics — which can be applied. We shall built these in as defaults.

- If the format of specifications is left sufficiently flexible, it will be open to the user of the system to override the defaults.

An example will illustrate these points.

1. Suppose we specify a total disk storage capacity of 3.3Mb. We could achieve this by having seven 7937H model disk drives, or by having nine 7933H disk drives. The former would be better by the rule of thumb which says "Fewer,

higher capacity disk drives are often better than more, lower capacity disk drives". Both would be acceptable, but we would hope that the first solution would be returned in preference to the second.[4]

2. In *some* circumstances we might want more disk drives (of smaller capacity). This could be because we wanted to increase the number of communication channels if our applications involved a lot of disk access. We can state this in the specification, for example by giving a minimum number of disk drives for the configuration; if this was (say) eight in the example, then the first solution will not be admissible but the second will.

3. There are many other solutions possible. However many of them would be considered poor design: for example, mixing different models of disk drives by having (say) four 7937H disk drives and three 7933H drives. But note that the user can still specify such (completely *legal*) combinations explicitly — this is the second specification option of Section 3.2.3.

3.6.4 Ensuring locally optimal solutions

Let us start by defining the concept of *utility*, which we shall encounter here in determining what, if anything, would constitute an optimum solution:

Definition 3.1 *The* utility *of a system is a multi-dimensional measure of its worth to the user. These dimensions may vary, but at present I shall enumerate them as*
(1) cheapness (inverse of cost)
(2) efficiency
(3) expandability
(4) anti-obsolescence (inverse of obsolescence)
(5) technological innovation

---

[4]The second would still be found, if more solutions are asked for.

These dimensions are named so that a move in the direction of the positive axis constitutes "improvement". However, I shall usually refer to *cost* rather than *cheapness*, and to *obsolescence* rather than *anti-obsolescence* in discussing configurations, since these are more natural.

Note that:

- *Obsolescence* is a property of *old* components which are likely to be discontinued (and therefore, ultimately, unsupported).
- *Technological innovation* is a property of *new* components in the process of becoming supported, which (it is hoped) represent the new "state-of-the-art" to come.

It is hard to address the question of what constitutes an *optimum* solution in a domain with several dimensions. On the one hand, we can *measure* the cost of a system: this is an objective, quantitative property. On the other hand, it is not true to say that we can *optimize* on cost considerations since cheaper solutions may be inferior in some respect to more expensive ones. We can talk about the efficiency of a system, but this is not so easily measurable. Also, two identical instances of configurations may differ in their utility, one running very efficiently and the other poorly — simply because they are running different applications, or are being used differently. In Section 3.6.2 we saw that there is a soft constraint to restrict the number of disk drives to fewer than the number given by the hard rule for the configuration. Suppose we have three identical configurations, each violating the rule. These configurations are part of *systems* (where a *system* includes the configuration, the software it is used to run, its environment, which includes the pattern of usage — interactive, batch, overnight, *etc.*) such that $\mathcal{C}_1$ runs inefficiently whereas $\mathcal{C}_2$ and $\mathcal{C}_3$ do not. How can this be? Suppose configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ both run a set of applications $\mathcal{S}_1$, and configuration $\mathcal{C}_3$ runs a different set of applications $\mathcal{S}_2$. $\mathcal{C}_1$ runs inefficiently, due to the fact that the applications it runs involve high disk access. It can be the case that $\mathcal{C}_3$ runs efficiently because $\mathcal{S}_2$ is not involved in much disk access. $\mathcal{C}_2$ may be acceptable because its high disk access applications are run overnight, and it does not matter if they run slowly.

Thus it is not possible to address the question of optimality at a global level, as is acknowledged in Frayman & Mittal (1987), who also address this issue. However, we wish to ensure that our solutions are at least *locally* optimal. I define this concept thus:

Definition 3.2 *A* locally optimal solution *is one whose utility cannot be increased (improved) along one dimension without a simultaneous decrease (deterioration) along at least one other dimension.*

We can compare solutions of equal cost, expandability, obsolescence, and technological innovation — probably solutions using an identical set of components but connected differently — along the dimension of efficiency, and ensure that we reach the one of the better solutions possible for that set of components. To this end, in the example of Section 3.6.1 we rejected the solution of placing the printer and the disk drive together because this was a less efficient solution, despite having the same cost as our favoured solution — note that it must necessarily have equal cost since it uses exactly the same components.

3.7 Soft constraint relaxation

The need to relax soft constraints arises when applying all of the soft constraints would leave the solution space completely empty. Unfortunately, constraints interact in such a way as to make this quite a difficult problem. Expert configurers tend to use rules of thumb based on experience coupled with their understanding of why the heuristic rules are there in the first place.

Let us give an example. Suppose we have four card cages and cannot configure any more, since this is the maximum allowable by the (hard) rules for this particular configuration. Suppose we have a total of twelve interface cards configured in these card cages. Of these, nine are HPIB channels. Suppose now that we have twenty-eight disk drives to configure.

We have, in this context:

- Hard constraints
    1. No more than five interface cards to a card cage
    2. No more than four disk drives to a channel
- Soft constraints
    1. No more than an average of three interface cards per card cage
    2. No more than three disk drives to a channel

These combine, given that we have four card cages, to:

- Hard constraints
    1. No more than seventeen HPIB interface cards in total
    2. No more than four disk drives to an HPIB
- Soft constraints
    1. No more than nine HPIB interface cards in total
    2. No more than three disk drives to an HPIB

While obeying both the soft constraints we can configure at most $3 \times 9 = 27$ disk drives. To configure the 28th we must relax *either* soft constraint (1) or soft constraint (2). It is not clear which would be preferable (or least unacceptable).

Relaxing constraint (1) will overload the card cages. Relaxing constraint (2) will overload one of the channels. Both of these have an effect on the efficiency of data transfer, in general. However, the actual effect of overloading in each of these ways depends on the particular extent and pattern of data transfer, and this in turn depends on three factors:

- The amount of disk access.
- The timing of disk access.
- The distribution of the data across disk drives.

So, for example, we would be prepared to relax constraint (2) if disk access were comparatively infrequent for the disk drives involved.

Soft constraint (1) is interesting. It was given to me in the course of knowledge elicitation but turned out to be "compiled knowledge". It is an example of a grossly over-simplified (in fact distorted) version of a rule given by people who *do* understand configuration to those who they believe are not capable of such understanding: the idea being that if the sales representatives blindly follow this rule it will at least keep them out of trouble. Its real *raison d'etre* is that the engineer who installs the configuration should be free to permute channels between card cages for maximum efficiency. It has not been possible, so far, to capture this knowledge, and indeed there are not many cases where it is needed. It is true that it is desirable, where possible, to allow the engineer who actually installs the configuration this flexibility, so that she can place interface cards in the optimum positions for efficient data transfer. However, there are certain patterns of system use which render it irrelevant, and therefore a constraint which can be readily relaxed. The motivation behind this heuristic is explained as: if there are not too many interface cards jockeying for position, then it is more likely that we can ensure that there are not too many of a particular type on one card cage. If the average rises above a certain level, then it is less likely that this can be done. However, by this formula the precondition for applying the heuristic has been lost. This seems to be a common problem in knowledge elicitation for AI, and for expert systems in particular.

3.8 Design issues

Much guidance in searching for a configuration can be provided if the user of the system has facilities for scene setting; *e.g.* stating whether an efficient configuration is important, whether there is a cost constraint, *etc.* There is a small subclass of design problems in which *the* 'optimum solution' (along one of several possible different dimensions: cost, efficiency ...) which meets the specifications is required. The underlying strategy to be used in this design scenario is quite different from, say, that to be used if the problem is to find any satisfactory solution(s) subject to keeping under a cost ceiling. As well as cost and efficiency, other

design issues are *expandability*, or potential growth path, of a configuration, *i.e.* its capacity to be augmented at a future date; and technological issues, such as whether certain hardware is obsolescent and will be superseded in the near future; and anticipating new trends in hardware.

3.8.1 Constraint management and optimization

I believe that synthesizing computer configurations is a question of *design*, where some human intervention is necessary. At the very least, with the minimal human input, the user should be provided with a set of solutions to choose from.

Not everyone sees the problem this way. Some assumptions that are commonly made are

1. That the problem can be viewed best as one of constraint management.

2. That the problem can be seen as one of optimization.

Other systems, such as that described in Frayman & Mittal (1987) use the constraints to "drive" the inference process in a MOLGEN-like system. However, when I examined the rule-base for the configurations I was considering, I found that they were not constrained to anything like the extent of the configurations described in Frayman & Mittal (1987), or, for that matter, in those of McDermott (1982), Soloway et al. (1997), Bachant & Soloway (1989), and Barker & O'Connor (1989). In all realistic examples, the solution space was very large, if I took *legality* of configuration alone into account. Applying soft constraints could prune the search tree, but could result in no solutions. Heuristic constraint management is a complex process, especially if the motivation behind the heuristics is ill-understood. In a typical negotiation between a computer hardware firm and a customer, the firm aims to provide a solution, or a small number of alternatives, which both meets the user requirements and offers competitively priced facilities, including the hope that the configured system will run the users' software in a reliable and efficient manner.

Let us examine criteria by which a solution may be judged good or bad, and assess what optimality means

- for each criterion separately
- for *all* criteria considered together

3.8.2 Cost

Sometimes a ceiling is placed on what may be spent on a configuration, in which case this forms part of the specification and represents a hard constraint on the solution space. Even if there is no such explicit constraint, I would not usually regard cost to be completely unimportant.

It is often assumed that minimising cost is a desirable goal. This is something of an over-generalization because

1. Often cost is of little concern, provided the budget ceiling is not exceeded.
2. It may be desirable to *meet* (exactly, if possible) the budget ceiling, if there is one. Often capital which is allocated but not spent is not transferable, so that the department concerned loses out; and also risks its budget being cut for the next financial period.
3. In the common situation of tendering for a contract, it is of course important to offer a competitively-priced configuration, but not at the cost (necessarily) of cutting corners.

At best, therefore, I can *sometimes* see the need for a strategy being designed to achieve the "cheapest, all other factors being equal" configuration. I move on to a discussion of these "other factors".

3.8.3 Efficiency

Sometimes a configuration, while legal according to the rules in the object level knowledge base, does not run very efficiently in practice, because of the way in

which the components are connected. For example, it may take a long time to transfer data to and from disk, or to print out documents, or to run compute-bound programs, or all of these. This issue was explained in Section 3.7. Additionally, a configuration may be more or less efficient, according to the choice of devices. For example, a disk drive with cache memory is faster than one without.

To appreciate the problems in trying to find 'optimum' solutions, there are a number of related points to be made.

- A configuration is *potentially inefficient* if the *heuristic* maximum (of disk drives per channel, of interface cards per card cage) is exceeded.

- If it is possible to breach the soft limits by varying amounts, then one assumes that the greater the breach, or number of breaches, the more inefficient the configuration, but...

- ...Provided the soft limits are *not* breached, a configuration is not made *more* efficient by further decreasing the number of components involved.

- Given a choice between breaching one soft limit or another, in the absence of evidence to the contrary we cannot *prefer* breaching one to breaching another; this is a genuine choice point in the search for an efficient configuration.

- Further, we cannot necessarily say that two breaches of one soft limit is worse than one breach of another.

- It *may* be the case that we would want a strategy for ensuring the least total number of breaches; however we might want it to be possible to return other solutions as well. The same applies to other strategies: ensuring the least possible breaches of soft limit A, or ensuring the least possible breaches of soft limit B, *etc.*

What seems to be desirable is that there is some means of capturing each alternative strategy, a search procedure which gives alternative solutions based on

each. Ideally we might want to allow some kind of interactive facility for choosing between them. We shall return to this in Chapter 8.

3.8.4 Reasoning with cost constraints

Suppose we have a cost constraint. Our strategy can be summarized as:

- If we can obey all soft limits, then we shall do so.
- If not, then we would want configuration with *maximum utility* (see Definition 3.1) subject to the cost constraints.

1. If there are two solutions which are within cost but each breaks (say) one heuristic — a different one in each case — *someone* needs to decide which of the two is acceptable/better.
2. Two solutions, where one has two breaches of soft limit A and the other has one breach of soft limit B, may not be comparable.
3. We may have to decide, say, between having faster (and more expensive) disk drives which are connected inefficiently, and slower disk drives which can be connected according to soft limits.

3.8.5 Reasoning without cost constraints

Just because we have no cost constraint does not mean that cost is unimportant.

1. It may be that we have to break soft limits designed to *ensure* efficient configurations anyway, because of the sheer size of the configuration. Then the same reasoning as in Section 3.8.4 holds.
2. If this is not the case, then a solution which does not breach soft limits may be found. However, we may be interested in returning other solutions as well, or in finding a minimum cost solution, for the purposes of negotiation. This is particularly true in tendering situations.

3.8.6 The problem with metrics

I do not believe that it is possible to develop metrics in order to decide a global optimum configuration for a given specification. Let us examine the reasons for this conclusion.

It might be argued that we *can* attach metrics (or measures) of how 'bad' a given breach is. Then the strategy should be to find an 'optimum solution', which mimimizes the 'total score'. There are several problems with this.

- It is unlikely that an expert configurer could give figures which would hold across all situations. How 'bad' it is to overload a channel with disk drives depends on the pattern of disk access, which depends on users, applications, and general environmental factors for the system.

- It is equally difficult to compare different 'bad' things. Sometimes we can put up with overloaded card cages but not with overloaded channels; sometimes it is the other way round: it depends on our ideas about the likely pattern of data access/transfer.

This problem is well known from cost-benefit analysis, for example, Ball (1979) writes:

> "The official figure in Britain for the value of a life in the early 1970s was around £9000. The absurdity of such a number is readily apparent. A plane carrying 100 passengers is, on this evaluation, transporting a cargo 'worth' £900,000. The jet aircraft, on the other hand, has, say, a replacement value of £2m. Imagine that this aircraft, while airborne, suddenly develops mechanical failure. The pilot is faced with the choice of irreparable damage to the aircraft in order to save the passengers, or jettisoning the passengers in mid-flight in order to save the plane. The pilot consults the cost-benefit manual to find out what society would prefer. After safely landing the now lightened aircraft, the pilot explains the absence of passengers by reference to the mechanical difficulty and the cost-benefit manual. Heroes, it seems, can come in many different guises, particularly if you believe in cost-benefit analysis."

So I would argue against hard-wiring any optimality strategy into the system: instead I need several strategies for synthesizing configurations which may possibly

be less than efficient. Flexibility is the key. I will demonstrate how this flexibility can be achieved on Chapter 5.

3.9 Conclusion

Simply expressed, the task that I have set out to automate is that of synthesizing configurations which meet the specifications given. Search is constrained by the need to generate *legal* configurations, and by constraints which may be present in the specification, such as the need to keep within a budget.

There appear to be three facets to the task:

1. Capturing the hard knowledge that we have, *i.e.* the object-level theory of configuration. Finding solutions according to these rules ensures *legal* solutions.

2. Capturing heuristic knowledge. These are definite facts about what constitutes a *good* solution.

3. Experimenting with strategies which aim to return solutions which are *as good as possible.* This area is more fuzzy — hence the need to experiment.

Chapter 4 deals with the first of these, Chapter 5 describes how the other two may be used to guide search, Chapter 6 is an account of how these were implemented, and Chapter 7 gives the results of various experimentations with this implementation.

Chapter 4

Formalizing the domain

Mein Hut, der hat drei Ecken,
Drei Ecken hat mein Hut,
Und hat der nicht drei Ecken,
So ist es nicht mein Hut!

Anon.

4.1 Introduction

In Chapters 2 and 3, I defined the configuration task, albeit in an informal way, and looked at some software systems for carrying it and related tasks out. In this chapter I shall give a more formal account of the domain and task.

I begin, in Section 4.2, by motivating the use of logic which is central to my approach. In Section 4.3, I define what is meant by a *legal* configuration and show how theorem-proving is relevant to the task by analogy with program synthesis. I shall explain how configurations are synthesized via proofs.

In Section 4.4, I define some terms and motivate both the choice of atomic objects in the theory and the method of building up more complex structures from simpler ones. The key factors here are maintainability of the system, and the need to deal with connectivity constraints.

Every object in the theory belongs to a *type*. Sections 4.5 and 4.6 discuss the types used in Section 4.7 to restrict configuration objects. Section 4.8 gives all

the axioms and rules of the theory and Section 4.9 indicates how they are used in proofs. Section 4.10 discusses the search problems involved with finding proofs and suggests meta-level techniques to deal with them. Appendix B.2 gives a complete list of all types and tactics implemented in the system.

4.2 Advantages of logical formalism

I have developed an axiomization of the theory of configuration which allows us to make use of standard theorem proving techniques. In particular, we view the synthesis of computer hardware as being analogous to program synthesis, which I describe in Section 4.3.

The first issue to be addressed is that of representing the configuration domain in a logical formalism. This is desirable in order to make it more precise, to enable reasoning about the domain in a rigorous manner, and to ensure soundness.

As we might have expected, this has proven to be a difficult task. The knowledge used by experts in performing their tasks is often presented in a haphazard and informal manner. Moreover, it is not easy to impose order on the plethora of objects at our disposal, or to arrive at more than an *ad hoc* description of how different objects may be combined.

However, the benefits of representing the domain as a sound logical theory far outweigh the difficulties. If we can assign a *type* to every object which is used in configuration problems, then the task of ensuring that various constraints are satisfied within a configuration is made much more elegant. If we can develop an axiomization of the rules concerned with the configuration task then the tasks associated with configuration can be seen as by-products of theorem proving in the domain, with the soundness associated with such techniques.

Within this theory, we may synthesize configurations c obeying certain requirements $spec(c)$ via proof of the theorem

$$\exists c.spec(c)$$

as described in Section 4.3.

This chapter describes the extrication of the object-level knowledge, in the form of hard facts and rules about computer hardware and its configuration, from the wealth of knowledge brought to bear by experts when carrying out configuration tasks; and covers the formalization of the object-level knowledge. Whereas the explicit representation of strategies for configuration may be developed by questioning experts, by reading configuration manuals, and by other knowledge elicitation techniques, the logical representation of the object level presents additional problems associated with the formalization of an originally non-formal domain. This is the crucial stage of developing an automated process. On it depends the soundness of the resulting knowledge based system.

4.3 Representation

A configuration consists of many components, which are connected together in such a way as to constitute a legal, working configuration. There are several basic types of component: for example, *example_printer* might be the name of a particular model of a printer; *some_terminal* a model of terminal, and *some_cable* a model of RS232 serial cable.

My technique is analogous to the program synthesis technique of, for example, Bates & Constable (1985) where we are given a specification $spec(input, output)$ of the relationship between the input and output, and synthesize an algorithm alg as follows:

> Find a *constructive proof* of the theorem:
>
> $$\vdash \forall input.\exists output.spec(input, output)$$
>
> Synthesize from this the algorithm alg such that
>
> $$\vdash \forall input.spec(input, alg(input))$$

By a *constructive proof*, we mean that it is not sufficient to prove existence, but the proof must produce a witness — *i.e.* an actual instantiation — for the existential variable *output*. As an effect of this process, we can also extract from the proof a procedure for constructing this witness.

For example, to synthesize an algorithm to sort a list of objects of type τ into ascending order we prove the theorem which states that, for all lists l_i, where l_i is a list of objects of type τ, there exists a sorted list l_o such that l_o is a permutation of l_i. The specification for such an algorithm is

$$\forall l_i{:}\tau \exists l_o{:}\tau.perm(l_i, l_o) \wedge ord(l_o)$$

where $ord(l_o)$ means that l_o is ordered. The extracted algorithm alg sorts l_i into l_o, for any such l_i, and provides the means of constructing *output* from *input*.

We aim to synthesize a hardware configuration in like manner, as a by-product of a proof. Suppose we are given a specification, $spec(c)$, for a computer hardware configuration. This specification will lay down certain conditions that the desired configuration must fulfill; for instance, that it has disk storage capacity of at least 1.2Gb; that it has certain printers; that it can support a given number of terminals running certain applications; and so on.

A computer configuration c which meets the requirements of a specification $spec(c)$ may be synthesized as a by-product of proving the theorem[1]

$$\exists c{:}\textbf{configuration}\ .spec(c) \tag{4.1}$$

Section 4.7 explains in detail what it means for a configuration to be well-formed, but a brief description is given here.

To be well-formed, a configuration object must possess certain properties, which may be summarized as

1. Certain components are essential (processor, memory, ...)

[1]We use the convention that types are represented in **bold face**.

2. All devices (disk drives, tape drives, printers, ...) must be connected into the configuration in some (legal) way.

Note that an alternative way of thinking of 4.1 is to introduce a meta-variable C and to prove

$$spec(C) \tag{4.2}$$

In proving Theorem (4.2) above, where $spec(C)$ is, in general, a conjunct of goals expressing the required properties of the configuration, C is instantiated to a well-formed computer configuration term.

This is a gradual process. Initially, we set out to prove

$$spec(C)$$

(Equation 4.2) where C is a meta-variable. In the early stages of the proof, C will acquire some structure, *e.g.*

$$C \equiv proc{:}L$$

where *proc* is instantiated but L is not: read this as "C consists of a processor list *proc* **and** some other terms". Later in the proof L acquires some structure, maybe a list in which one or more elements are instantiated, and so on.

4.4 Objects of the theory

4.4.1 Motivation for types

First note that my particular choice of atomic types enables me to formalize the expression of an important class of constraints in this domain, namely those concerned with ensuring legal connections. We wish to reduce this kind of constraint-checking to well-formedness checking, embedded in the logical representation of the configuration domain.

Computer hardware configurations are gradually built up by taking individual (simple) components and building them into more complex (compound) structures.

Mirroring this, I shall start with a description of how simple components may be represented, and how these objects may be combined to give members of compound types. A computer hardware configuration is then represented as an object of type

$$list(\textbf{processor}) \times list(\textbf{memory}) \times \ldots \times list(\textbf{terminal}) \times list(\textbf{disk}) \times \ldots$$

where we can read $\times$ as "and" — in other words, a configuration is composed of a processor (list) *and* some memory *and* consoles *and* terminals *and* disks *and* ...

The configuration object will arise out of the proof of a theorem of the form given in Theorem 4.1, part of which is the proof of well-formedness of witnesses to the existential goals. This means that any object c which appears as an instantiation of a **configuration** object is guaranteed to be a legally constituted computer hardware configuration.

Remember that we defined the term *utility* in Chapter 3 (Definition 3.1). Let us now define the term *usage* as follows:

Definition 4.1 *The* usage *of a component (of computer hardware), or group of components, is its function, in terms of the service it provides to the user of the computer configuration. A component may have more than one usage.*

An example of a component having more than one usage is a tape drive, whose usages are *backup of data* and *software transfer*. In Chapter 3, I discussed the problems involved in classifying objects in a hierarchical way. Further examples of this are:

- We can classify according to *usage*, defined above; for example 7937H(3) and 7937FL(1) are disks; 2932A(5) is a printer.

- We can classify according to *connectivity* within the configuration, for example 7937FL(3) must be a fibre-optically connected device; 7937H(2) must be a *system device*, by which I mean that it must be connected via an object of type **system_cable**; 2932A(6) may be either a system or a *serial device* (connected via an object of type **serial_cable**).

- A PC may be free-standing or considered as a terminal in a large networked system.

- Disk storage may be provided in a number of ways in different configurations, for example:

  - by means of disk drives connected to channels via a cable
  - some models of mini-computer have built-in storage (the configuration of which they are part may have separate disk drives as well)
  - there are devices which combine the functions of disk and tape storage
  - there may be devices providing corresponding usage but not classified as disk drives.

I have chosen a fairly flat classification for the reasons given in Sections 4.4.2–4.4.5. By *flat* we mean that there is little use made of any hierarchical structure in the static representation. Thus we do not, as is the case in frame-based expert systems, use inheritance of properties. We shall later see the use of *union types*, but this will be simply to allow lists of differently typed objects to be formed in certain circumstances (see Section 4.6.2).

4.4.2 Maintenance

Maintenance is a problem for XCON and similar systems (Chapter 2, Section 2.3.2). It would be good to have a system which could be maintained by people who did not necessarily have knowledge *of the system as a whole.* In other words, someone whose job it was to maintain the price and product list as new components came into production and obsolete ones were deleted would not have to know anything other than how to maintain the particular files which hold the information relevant to this task. Similarly, an engineer should be able to maintain the part of the system requiring her particular expertise independently of the rest of the system.

If we adopted a hierarchical structure we would constantly have the problem of fitting in new devices to this rigid framework. It is a common problem in Artificial

Intelligence that initial classifications within frame-based systems and the like break down when new objects are introduced which defy the original classifications, or if the information is put to a different use. For instance, if we attach the property "flight" to the class of birds because it is useful in many cases we need somehow to cope with exceptions — the ostrich or the overweight turkey. This leads to rethinking either the classification or the properties attached to slots or both; or else to messy exception-handling procedures. The problem in the computer hardware domain is that we cannot predict the course that technology will take. New products might cut across existing divisions: maintenance of the system would mean not simply updating the product data but also maintaining the structure. This would add an unnecessary overhead on to an already onerous task. My aim is to make maintenance as straightforward as possible. The knowledge base part of the system can be updated by people who currently maintain product information — people who do not necessarily have the expertise needed to maintain a structure tree for the knowledge base; so that unless all future products conform to the existing structure the addition of just one "revolutionary" component will cause problems.

4.4.3 Separation of usage and connectivity

Each component has two important attributes:

1. its value (the user view)
2. its means of connection (the engineer view)

Let me now define the term *value*, in terms of *usage* as previously defined (Section 4.4.1, Definition 4.1):

Definition 4.2 *The* value *of a component, or group of components, is the set of usages, paired where appropriate with a measure of how well each function is performed. This measure may be quantitative or qualitative.*

The value of a component, group of components, or complete configuration, to a user is, put simply, a description of what it does for the user, and how well.

The value of a printer is the printing services it provides — for example, the fact that it offers double-sided printing, and the speed at which it prints. This speed is probably measured quantitatively, for example "600 lines per minute". The value of a group of disk drives is the total capacity of the storage they provide, and maybe the rate at which data can be transferred to and from them. The latter may well be measured qualitatively — *e.g.* "fast". The value of a complete configuration is the description of all the facilities it provides — printing, disk storage, the applications which can be run, the numbers of terminals of various kinds, and so on. The desired value of a configuration is given in the specification — although the actual value may exceed this specification.

4.4.4 Connectivity

The engineering view of a component is not usually in terms of what it does but how it is connected. So the interesting things about disk drives and printers are the options for connecting them. Often there is only one option: then the question is simply whether they can be connected (whether there are sufficient resources, in terms of vacant sockets *etc.* or not). If there are apparent alternatives, then the choice made may have repercussions later, for example by consuming resources needed by other components which cannot be connected in any other way. Occasionally the choice of connection method affects the value of a component. For example, connecting a printer by means of a system cable means that it cannot be used remotely, but must stand fairly near the processor, probably in the same room. It is likely that in these cases, where it is important, the choice of connection method will feature in the specification.

With this caveat, usage and connectivity factors are, in general, independent:

1. Not all devices with usage *printing* will be serially connected.

2. Serially connected devices between them provide a number of usages, *e.g.* printing, communication.

The problem is that at different stages of the synthesis, we are interested in different attributes; it is also true that different people are interested in different things and we must preserve both views. For example, the focus changes during the course of synthesizing the configuration. At the start, the main focus is on the user view, and the value of the configuration in terms of what services it provides (printing, applications, disk storage). We consider devices in terms of their value, and how they are combined to provide the value specified by the user requirement. Of course, this may well have implications for how some of the devices are connected. Later, I shall focus on the the details of how *all* the various devices may be connected, which is the engineer's view.

Because two views exist, we should need to maintain not one, but two hierarchies. This is feasible, but leads to the problems outlined in Section 4.4.2, doubled.

4.4.5 Connection constraints as types

Whatever architecture is used for the configuration system, it is common practice to store attributes of components relating to their value in a knowledge base. In following this practice I wish, for the reasons hinted at in the previous section, and given more fully in Chapter 5, Section 5.2.1, to keep the attributes relating to how the components are connected separate from other attributes. The former is normally of no direct interest to the user, even though in some cases how a device is connected can affect its value. So although my system will later need to incorporate some heuristic meta-level knowledge relating method of connection to efficiency of operation, expandability, or cost, this must not appear at the object-level, where I merely wish to ensure that the requirements for legality are met.

The knowledge about how components are connected is constraint information. We can make a general statement to the effect that all devices must be connected by some path to the rest of the system. This is axiomatic to any general theory. Specific examples of paths are cables, modem, *etc.* Some systems have card cages with slots for channels, to which the other end of cables are attached. Note that the general theory does *not* say that we can use *any* cable or *any* slot. The knowledge

about which objects are acceptable is a constraint on the configuration synthesis. It is embedded in the type theory.

These constraints only come into play when we are trying to combine two or more components. I have chosen to embed them in the type definitions. Constructing a compound object, such as a device and its associated cable (*i.e.* a member of the type **device**×**cable**) or a device×cable pair with a slot in a card cage, is seen as constructing a new (compound) object. Whether or not this can be done, legally, reduces to whether the object so-formed is in a type — *i.e.* is well-typed. For example, the function

$$connected\_via : \textbf{device} \times \textbf{configuration} \mapsto \textbf{channel}$$

which gives an output *ch* of type **channel**, requires two inputs: the second of these is a **configuration** object *c* (maybe partially instantiated); and the first must be an object *dev* in **device**, and

$$\exists cb{:}\textbf{cable}.connect\_cable(dev) = cb\ in\ c$$

i.e. there is a cable, *cb*, connecting *dev* in the configuration *c*.

We shall write

$$connected\_via(dev) = ch\ in\ c$$

Introducing too much super/sub-type structure into this theory is distracting and unnecessary. Since there would be so many exceptions we cannot reap the usual benefits of this kind of representation — *viz.* inheritance of properties — without wrecking the clean logical framework I am attempting to build. This seems to justify aiming for the flattest possible representation, and rather building hierarchical structures *dynamically.* This dynamic building of structures avoids the need for over-constraining static hierarchies.

4.5 Simple types

4.5.1 Components as members of types

We represent individual components such as processors, memory modules, terminals, disk drives, tape drives, printers, channels, fibre-optic links, and cables as members of atomic types (sometimes referred to as *sorts*). For example, *hpib*(5) is a channel component, *i.e.* an object of type **channel**. We can readily add new types of components to this representation, as shown in Section 4.6.1.

The simple types we will come across in configuring large mainframe systems are described in the sections which follow.

4.5.2 Devices

Devices include disk drives, tape drives, terminals, printers, *etc.*, with respective types **disk**, **tape**, **terminal**, **printer**. The type **disk** is a union type over "ordinary" disk drives and those connected via fibre-optic links. Union types are explained in Section 4.6.2 below.

4.5.3 Channels

HP 3000 computer systems use various kinds of channel for connecting devices. These are MUXes, LANICs, HPIBs, and HPFLs (fibre optic links), of types **mux**, **lanic**, **channel**, and **fchannel**.

4.5.4 Other connecting components

To connect terminals, apart from the system console which will be connected via a MUX, we have DTCs, portgroups (for cables or modems), cables (RS232 and RS422), and modems.

The function of all the above components can be found in the Glossary Of Hardware Terms and the type names in Appendix D.1.1.

4.5.5 Miscellaneous

Other important types are **processor**, **memory** (modules), and **cardcages**. Each processor will have particular models of memory board associated — sometimes there is only one but sometimes differently sized memory boards are available. Some processors have the option of configuring extra card cages to allow more devices to be configured whilst others are fixed in this respect.

4.6 Compound types

More complex types can be composed via *type constructors*. Type constructors are needed to select members of a type (functions), to combine objects of the same type together in lists with corporate properties (parametric lists), and to describe component connections (cartesian products). These are described in this section.

4.6.1 Function types

Attributes of devices are not associated with individual instances of components, but with *models* of devices. So, for example, the 7937XL is a model of disk drive which has cache memory and a storage capacity of 571 megabytes. There may be several instances of such disk drives in a configuration, and we need to refer to these somehow. The neatest way is to index the model name, as follows.

Suppose we have a model of some component, *model_name*. Suppose the type of such components is τ. Then we name a function *model_name* which maps members of the natural numbers $\mathcal{N}$ to members of τ:

$$model\_name : \mathcal{N} \mapsto \tau$$

For example, $7937XL$ is a model of disk drive.

$$7937XL : \mathcal{N} \mapsto \textbf{disk}$$

is a function, and

$$7937XL(1),\ 7937XL(2),\ 7937XL(3),\ \ldots$$

are instances of this kind of disk drive in a configuration, each of them having identical properties (cache memory, storage capacity 571Mb, *etc*).

When we want to add new models of component to the knowledge base, we can do so by adding a new function to the file of component models together with the type of its codomain[2] (its domain is always $\mathcal{N}$), and inserting its attributes of objects in its image into the attributes file. For example $7937XL$ has codomain **disk**; objects of the form $7937XL(n)$, with $n \in \mathcal{N}$, have capacity 571Mb, *etc.*

Adding a new *type* of object is also straightforward. Suppose a new type of storage device is invented. We define the type, **storetype**, say. Suppose we have two different models of **storetype** about to start production, called $fast$, and $whizzo$. We incorporate the functions

$$fast : \mathcal{N} \mapsto \textbf{storetype},$$
$$whizzo : \mathcal{N} \mapsto \textbf{storetype},$$

Then objects of the form $fast(n)$ or $whizzo(n)$ will be instances of type **storetype**.

In addition, we need to define the well-formed types involving members of the new component type, to determine how these may be connected into configurations (see 4.6.3). For example, if all members of **storetype** can only be connected via an object of type **widget**, then we define the cartesian product type **storetype** $\times\ \tau$ to be well-typed only if $\tau =$ **widget**.

---

[2]otherwise known as its *range*

4.6.2 List types

If τ is a type, then $list(\tau)$ is a type whose members are lists of members of τ.

- We define the list of no members as *nil*
- The constructor function is *cons*:

$$cons : \tau \times list(\tau) \mapsto list(\tau)$$

  which takes an object a of type τ and a list l of objects of type τ and returns a list of objects of type τ consisting of the object a followed by the elements of l, in the order they occurred in l. We write this $a :: l$.

Frequently we wish to refer to a list of devices of the same type. For example, we may wish to calculate the total storage capacity of the disk drives in a configuration. If we can group devices together, such as the disk drives of a configuration, then composite attributes such as capacity may be defined recursively over lists. For example, the partial configuration representing the disk storage devices might be

$$[example\_ddrive(1), example\_ddrive(2), another\_ddrive(2)],$$

of type $list(\textbf{disk})$. We can return the capacity of such a list in the way shown in Section 4.6.4.

We may also wish to define lists whose elements may be different types of storage device. *Union types* may be defined; for example, a union type over the different storage types. Then lists may be defined, each of whose elements is a member of the union type. For example we might have a list of devices connected to a given channel returned as

$$[example\_ddrive\_1(1), example\_ddrive\_1(4), example\_tape\_3(2)],$$

of type $list(\textbf{device})$, where, say

$$\textbf{device} \equiv \textbf{disk} \cup \textbf{tape} \cup \textbf{printer} \cup \textbf{terminal}$$

4.6.3 Cartesian product types

Connecting components together is represented by forming cartesian products of the two elements to be combined.

Cartesian products are defined as follows.

If **A** and **B** are types, then **A**$\times$ **B** is a type whose members are pairs $\langle a, b \rangle$, with a in **A** and b in **B**.

For example,

$$\langle printer\_1(5), system\text{-}cable\_1(16) \rangle$$

represents a printer with a system cable attached.

Not all components may be so combined: this restriction represents a significant class of constraints in the system. Thus we have dependent product types,

$$x : \mathbf{A} \times \mathbf{B},$$

where x is in **A** and **B** depends on x.

For example,

example_ddrive and *another_ddrive* are both functions into type **disk**,

but *example_ddrive*(3) requires a *system-cable*, whereas *another_ddrive*(2) needs a *fibre-optic-link*.

So if $x = example\_ddrive(3)$ we can form

$$\langle example\_ddrive(3), system\text{-}cable\_1(14) \rangle$$

as a well-typed object, but not if $x = another\_ddrive(2)$, since

$$\langle another\_ddrive(3), system\text{-}cable\_1(14) \rangle$$

would not be well-typed because *system-cable_1*(14) is not a fibre-optic link.

4.6.4 Arithmetic and list functions

In the process of synthesis, we will need to use various simple functions such as the length of a list, the sum of two natural numbers, *etc.*

Length of a list

For a list of objects of type τ we have *length*: $list(\tau) \mapsto \mathcal{N}$,

This is needed for checking constraints on (say) the number of devices of a particular type which may be connected to a given channel. For example, we may insist that

> if $devices(ch)$ is the list of all devices connected to channel ch
>
> and $devices(hpib(n)) = device\_list$:list(**device**)
>
> (where $n \in \mathcal{N}$)
>
> then $length(device\_list) \leq 6$, say.

Arithmetic

We need functions such as

> $plus$: $\mathcal{N} \times \mathcal{N} \mapsto \mathcal{N}$

This is needed, for example, in the recursive definition of capacity: the capacity of a list of devices is the capacity of the head plus the total capacity of the tail of the list:

> $lcapacity(nil) = 0$
>
> $lcapacity(a :: l) = capacity(a) + lcapacity(l)$

where $capacity(a)$ is the capacity of the component a.

4.6.5 Building more complex types

We can combine these constructors to build still more complex types. This culminates in the **configuration** type, whose members are computer hardware configurations.

For example, we can form the cartesian product of device and its connecting cable, as in

$$\langle example\_ddrive(5), system\_cable\_1(7) \rangle$$

We can form a list of such pairs to represent all the devices connected via their cables to a given channel, as in

$$\langle channel\_1(1), [\langle ddrive\_3(5), system\_cable\_1(7) \rangle, \langle tape\_2(1), system\_cable\_1(8) \rangle] \rangle$$

Finally, a configuration object is a cartesian product of lists

$$list(\textbf{processor}) \times list(\textbf{memory}) \times list(\textbf{device})$$

$$\times list(\textbf{connectors}) \times list(\textbf{connections})$$

4.6.6 Destructor functions

We often need to pull apart terms which have been constructed as outlined above. For example, the function *connected-via* takes a term of type

$$\textbf{configuration} \times \textbf{printer},$$

and returns the channel to which the device is connected in a given configuration via its cable:

$$connected\text{-}via : \textbf{configuration} \times \textbf{printer} \mapsto channel$$

Then we may want to refer to a particular group of components of a configuration c, for example:

$$disks(c)$$

returns the list of disk drives of c.

4.7 Configurations

Definition 4.3 *A component, comp, has a* valid *connection if the function connected_via(comp, c) is defined and returns a value, i.e.*

$$\exists \tau \exists x{:}\tau.connected\_via(comp, c) = x$$

I have defined the set of essential parts, referred to in the definitions, as

{processor, memory, system_storage, backup, console}.

I give here the definition of *legal*, as applied to configurations:

Definition 4.4 *A configuration c is* legal *if*
(i) *There are instantiations for the values of all functions taking the form* essential_component*(c), where* essential_component *stands in turn for each of the members of a designated set of essential parts.*
(ii) *Every component object of* c *has a valid connection.*

Recall that the syntactic structure of a configuration is a tuple

configuration = *list*(**processor**) × *list*(**memory**) × ...

The number of elements in this list, as well as their types, varies from configuration to configuration — it is dependent on the processor. Since all computer configurations have a processor and some memory the template fragment

proc:**list**(**processor**) :: (*m*:**list**(**memory**) :: *L*),

where L is a meta-variable standing for the rest of the configuration, will be the same for configuration, no matter what the processor is.

If we know, for instance, that the processor list contains a single HP3000 series 950 model processor (only), then we can expand this template to

$list(\textbf{processor}) \times list(\textbf{memory}) \times list(\textbf{terminal})$

$\times list(\textbf{terminal})^3 \times list(\textbf{disk}) \times list(\textbf{tape})$

$\times list(\textbf{printer}) \times list(\textbf{mux}) \times list(\textbf{lanic})$

$\times list(\textbf{channel}) \times list(\textbf{fchannel}) \times list(\textbf{cardcage\_term})$

A configuration, to be well-formed, must also obey certain properties, $prop_i, i \in \mathcal{N}$:

$$\forall c{:}\textbf{configuration}. \bigwedge_{1 \leq i \leq n} prop_i(c) \tag{4.3}$$

where these properties include, for some configuration object c:

- $processor(c) \neq nil$

  $\wedge memory(c) \neq nil$

  $\wedge console(c) \neq nil$

  $\ldots$ *etc.*

  which expresses the fact that certain components are essential

- $\forall x{:}disk \in disks(c).\ \exists y{:}system\text{-}cable \in system\text{-}cables(c).\ connect\text{-}cable(x) = y\ in\ c$

  where $x \in disks(c)$ means "x is a member of the list $disks(c)$", and similarly for y, which is in the list $systems\text{-}cables(c)$.

- $\forall x \in devices(c).\ \exists z \in channels(c).$

  $$connected\text{-}via(x) = z\ in\ c \wedge \exists cb.connect\_cable(x) = cb\ in\ c$$

  expressing the fact that all devices must be connected into the configuration via some means (cable, modem, *etc.*).

- $\forall x \in channels(c).\ \exists cc{:}cardcage \in cardcages(c).connected\text{-}via(x) = cc\ in\ c$

  expressing the the fact that each interface card must be configured in a card cage

- Conjuncts to do with constraints of either of two classes:

  1. Limits on the number of devices which can be connected to a channel or on the number of interface cards which can be configured in a card cage.

---

[3]There are two lists of terminals; the first are consoles (usually only one) for system use and the second is the list of user-terminals.

2. A small number of "incompatibility constraints", to do with the fact that certain devices, or certain devices fulfilling certain functions, may not share a channel.

A term of type **configuration** is a tuple consisting of a processor list[4], a list of system disks, a list of memory modules, lists of all the major components appearing in the configuration, and a list of card cage terms. Note that there will generally be constraints on the lengths of lists, determined by the processor type, which also determines which devices and other components are configurable (they must all be "supported" by the processor).

Note also that this representation of configurations appears to contain redundancy, in that the name of a tape drive, for instance, will appear twice: once within the list of tape drives and once in the card cage term. This reflects the need to reason separately about the value of a configuration in terms of its major components whilst proving high level specification goals; and the connectivity of the configuration, when proving well-formedness goals. Therefore this representation, with its judicious replication of names of certain objects, is ideally suited for the dual view of configurations and the configuration task depicted in Section 4.4.3. For a given object, each repetition of its name represents a different facet of its existence, and so is not redundant.

Syntactically, therefore, the definition of well-formedness for a configuration object includes its (syntactic) consistency; for example, that each device is present both in the list of card cage terms and in the appropriate device list (occurring exactly once in each).

To describe a configuration, we need firstly to be able to indicate its major devices. We use destructor functions for this, *e.g.* the terminals of a configuration c, $terminals(c)$:

$$terminals\text{:}\mathbf{configuration} \mapsto list(\mathbf{terminal})$$

---

[4]The reader might wonder why a list of processors is used, when in all examples considered here, the list consists of just one processor. However, in general we want to allow a *list* of *at least* one processor, to accommodate networked systems.

We may need to identify other components, for instances the channels via which devices are connected into the configuration:

channels:**configuration** $\mapsto$ *list*(**channel**)

We need to identify the connections present: which card cage a certain interface card is configured in:

connected-via : **configuration** $\times$ **channel** $\mapsto$ **cardcage**;

To identify the cable for a device:

connect-cable : **configuration** $\times$ **printer** $\mapsto$ **system-cable**;

to determine which channel a device is connected to (via its cable):

connected-via : *configuration* $\times$ (**printer** $\times$ **system-cable**) $\mapsto$ **channel**.

4.8 Axioms and rules

In the axiomization of the configuration theory, we must be careful to include only immutable rules, and to exclude heuristics which are used by human experts either to control their own search problems, or because they guide the search towards configurations which are desirable in some way. We cannot put these desirable properties in the object level as rules because they are heuristic only, and not universally obeyed: in the event of (say) cost constraints which prevented us achieving an efficient configuration, we would have to settle for a less efficient one. In my approach it is the task of the planning system to guide the search in this way at the meta-level.

I give some example axioms:

1. If the processor is known (or becomes instantiated in the course of the proof), then this provides a kind of skeleton or template for the configuration: for

instance, a certain minimum configuration must exist, consisting of (say) the processor, two card cages, a certain number of channels configured in one or other of the card cages. For each kind of processor *proc*, there will be a rule

$\forall c$:**configuration**.

$$template(c) \leftrightarrow$$
$$(c = [[proc], \ldots])$$
$$\wedge\ (cardcages(c) = [cc_1, \ldots])$$
$$\wedge\ (channels(c) = [ch_1, \ldots])$$
$$\wedge\ (connected\text{-}via(ch_1, cc_1)), \ldots$$

where the terms omitted (denoted by ...) are specific to the processor *proc*, associating each processor with its "template", or minimal configuration of card cages, channels, *etc.*, together with any known configuration details, such as which card cage each interface card goes in. These "minimal configuration" details are given by configuration manuals.

We will have one rule

$$template([[proc], \ldots]) \leftrightarrow (processors([[proc], \ldots]) = [proc]) \qquad (4.4)$$

for each processor *proc*. For example, the order number in the price list for a HP 3000 950 processor includes the processor itself, four 16Mb memory boards, two card cage adapters, one LANIC, two MUXes, and two HPIB channels.

2. There are also rules defining what it means for a device to be *connected via* a cable to a channel, *e.g.* for a disk drive *dd* and its cable *cb*, connected to channel *ch* in a configuration *c*:

   For *dd*:**disk**, *cb*: **system-cable**, *ch*:**channel**, *c*:**configuration** *dts*:*list*(**disk**$\times$ **cable**), *chtl*:*list*(**channel** $\times$ **device_list**)

   we have:

$$
\begin{aligned}
&\mathit{connected\text{-}via}(dd) = ch\ \mathit{in}\ c \leftrightarrow \\
&\quad dd \in \mathit{disks}(c) \\
&\quad \wedge\ \exists \langle \ldots, \mathit{chtl}, \ldots \rangle \in \mathit{cardcage\text{-}term}(c).\ \langle ch, \mathit{dts}, \ldots \rangle \in \mathit{chtl} \\
&\quad \wedge\ \langle dd, cb \rangle \in \mathit{dts}
\end{aligned}
$$

(...help to pick out the terms which are matched by this rule from the complicated structures involved)

where *chtl* represents the list of all channels in c together with, for each channel, the list of *device* $\times$ *cable* pairs connected to that channel; and *dts* is a list of such pairs.

Similarly, for an interface card to be configured in a card cage:

$$
\begin{aligned}
&\mathit{connected\text{-}via}(ch, c) = cc \leftarrow \\
&\quad cc \in \mathit{cardcages}(c) \wedge ch \in \mathit{channels}(c) \\
&\quad \wedge\ \langle cc, \ldots, \mathit{chtl}, \ldots \rangle \in \mathit{cardcage\text{-}term}(c) \\
&\quad \wedge\ \langle ch, \ldots \rangle \in \mathit{chtl}
\end{aligned}
$$

4.9 Synthesis proofs

I give here an outline of how synthesis proofs proceed in this paradigm.

Let us take a very simple specification (and therefore synthesize a somewhat unrealistic configuration). Suppose we want to have an *example_proc* processor (where *example_proc* is the name of a kind of processor) and disk storage capacity of at least 500Mb. We draw up the specification:

$$
\mathit{spec}(C) \equiv (\mathit{processor}(C) = \mathit{example\_proc}(N)) \wedge (\mathit{capacity}(\mathit{disks}(C)) \geq 500) \tag{4.5}
$$

Taking the specification theorem 4.1 together with the definition of what it means for a configuration to be well-typed (4.3), and with the specification given as in

4.5, we need to prove, for c:**configuration**:

$$(\exists c.(processor(c) = example\_proc(N)) \wedge (capacity(disks(c)) \geq 500) \wedge (\bigwedge_{1 \leq i \leq n} prop_i(c)) \tag{4.6}$$

The main highlights of the proof of conjecture 4.6 are as follows.

1. *The processor is synthesized.* Here the model of processor required is given explicitly in the specification. An instance of this model of processor is therefore selected.

   Axiom (4.4) matches the first conjunct of (4.5) with $proc = example\_proc(N)$, for some $N \in \mathcal{N}$. Say $N = 1$.

   $$template([example\_proc(1), \ldots]) \rightarrow processors([example\_proc(1), \ldots]) = proc$$

2. *Choosing the processor fixes the template,* as explained in Section 4.8.

   The goal $template(C) \ldots$ is proven by synthesizing the appropriate components to stand as witness for the existentially quantified variables in the subterms of C:

   $$template([example\_proc(1), \ldots) \leftarrow$$

   $$(C = [example\_proc(1), \ldots])$$

   $$\wedge (cardcages(C) = [\ldots])$$

   $$\wedge (channels(C) = [\ldots])$$

   $$\ldots$$

3. *The disk drives (actually only one) will be synthesized.*

   C is now a partially instantiated term, with its processor slot instantiated, and with partially instantiated lists in the card cage slots, channel slots, *etc.*:

$C = <[example\_proc(1), \_, [mem\_4(1), mem\_4(2)\|\_],$

$[cc\_1(1), cc\_1(2), \|\_],$

$\ldots,$

$[ch\_1(1), ...],$

$\ldots,$

$[< cc\_1(1), \ldots [< ch\_1, \_, \_, \_ >, \ldots], \ldots] >$

where mem_4 is a model of memory board of which $mem\_4(1)$ and $mem\_4(2)$ are instances; similarly cc_1 is a model of card cage, and ch_1 is a model of channel.

We turn our attention to the second conjunct of (4.5) and generate an object of type $list(disk)$ with the required property (*viz.* that the total capacity of the members of the list is at least 500. At this stage, the disk subterm of C is instantiated, say to $[example\_disk(1)\|\_]$, so that C becomes

$C =< [example\_proc(1), \_, [mem\_4(1), mem\_4(2)\|\_], [example\_disk(1)\|\_],$

$[cc\_1(1), cc\_1(2), \|\_],$

$\ldots,$

$[ch\_1(1), \ldots],$

$\ldots,$

$[< cc\_1(1), \ldots [< ch\_1, \_, \_, \_ >, \ldots], \ldots] >$

where $example\_disk(1)$ is some disk drive with a capacity of at least 500Mb.

4. *The appropriate connections are synthesized.*

The last stage is to ensure that $C = [example\_proc(1), \ldots]$ is a well-formed term. This is the major part of the proof and we use the conjunct referred to by 4.3:

$$\forall c{:}\textbf{configuration}. \bigwedge_{1 \leq i \leq n} prop_i(c).$$

This ensures that all the disk drives synthesized in step 3 are *connected via* suitable cables to channels.

At the end of this proof, C is fully-instantiated and of type **configuration**.

4.10 Search problems

Assuming that this axiomization represents a correct account of configuring HP 950 processors, we can, in theory, synthesize computer configurations using this object-level knowledge alone: indeed, simple configurations have been so synthesized. However, as one might expect, the search space is huge.

The results reported in Chapter 7 of applying the object-level theory alone on specifications show the problems of combinatorial explosion that arise from having a high branching rate and under-constrained solution space.

However, any solutions which do result (in cases where the search involved in finding them is not so great that none can be found at all) can be guaranteed to satisfy the specifications, and to represent legal configurations. So the system is sound.

The next stage, as we shall see in Chapter 5, is to utilize the control and heuristic knowledge which I have now carefully separated from the object level. I affirm that this methodology allows me to reinstate such meta-level knowledge in a principled manner.

To show the extent of the search problem, let us consider just a single specification goal. Typically, a goal might be "at least 2Gb of disk storage". There are arbitrarily many disk combinations which satisfy this goal. Also, while there may be a multitude of solutions, nothing at this level ensures that "good" configurations result; instead, we are likely to end up swamped by solutions which, whilst legal in the strict terms of obeying the rules, are inefficient in practice, or overly expensive, or generally undesirable. This is not the fault of my system but of the

specifications. A measure of the search problems involved is given in Chapter 6, Section 6.8.4.

In practice, specifications may not be fixed but "negotiable" — the goal posts keep moving. For example, suppose the expert configurer were to come up with a configuration which satisfies the specification, including a cost constraint, with a solution which is well under the cost specified. The customer may then ask for a more expensive solution, to use up the company budget allocation. There may be many ways of achieving this; for example by using more expensive (better?) components; or overspecifying certain goals.

Assuming that we have a rigorous axiomization which contains only the hard and fast rules for configuration, the way is clear to develop

1. Higher level *tactics* built up from the low level rules already implemented, which, for instance, will build partial configurations in response to conjuncts in the specification: *e.g.* the disk drives fully configured in the configuration.

2. A *method* for each tactic which specify when it is appropriate to apply the tactic, giving preconditions for the tactic and also the effect of executing the tactic. This methodology is explained in Bundy (1988).

3. The embodiment of heuristics in the methods: for instance, there could be a method for configuring devices so as to ensure an efficient computer configuration. A specification goal *efficient(c)* can be incorporated as a parameter to a *supermethod* (defined in Chapter 5, Section 5.5.3).

4. A planner capable of generating sequences of methods as plans which, if the corresponding tactics were executed, would synthesize configurations meeting the given specification. The search involved in planning is vastly less than that involved at the object-level, so that this alleviates the problems of combinatorial explosion.

4.11 Summary

The difficulties involved in maintaining large knowledge bases for configuration and order checking tasks are well known. An important factor in overcoming these is the clean separation of object-level facts from meta-level control and heuristic knowledge. I outlined how object-level knowledge may be extracted and formally represented in such a way as to allow the utilization of techniques analogous to program synthesis, to perform tasks such as synthesizing computer configurations which meet specifications. Such an approach makes the task of maintaining knowledge bases more tractable and reliable; the consequent separation of object-level knowledge from heuristics and control knowledge increases the usefulness of the knowledge base.

Many knowledge based systems suffer from the difficulty of separating object-level and meta-level knowledge. Knowledge bases are compiled with the aid of human experts who are often unable to unravel the two, or even realize the distinction. As new objects are added or new rules discovered it becomes increasingly difficult to maintain the system. Moreover, new applications, which ought to be based essentially on the same factual knowledge as the old one, cannot always plug into the existing knowledge base, since the information therein is irretrievably bound up with high level control knowledge used by the original application.

I have considered the tasks associated with computer hardware configuration. Although I have focussed on configuration synthesis, *i.e.* configuring computer configurations to meet given specifications, I have been aware that there are related tasks which use the same object-level knowledge: namely order checking (given an order, ensuring that a viable computer configuration can be built from it) and system upgrading (given an existing configuration, and new specifications, building a new configuration based on the old one). I have endeavoured to separate out and formalize this object-level knowledge so that it may be used by planning systems for any of these ends. As far as I know, little or no work has been done on

the last of these tasks, despite the existence of systems which successfully perform the first two, as outlined in Chapter 2.

Let us move on to Chapter 5, where I discuss the use of meta-level proof planning techniques, and to Chapter 6, which describes the implementation of object-level and meta-level theories.

Chapter 5

Proof Plans for Configuration

Dear God, ... I do not have my prayer book. And I have such a poor memory that I cannot recite the prayers by heart ... But You know all the prayers, Lord — so I'll just recite the letters of the alphabet, and You put them together in the right way.

from a Chasidic story

5.1 Techniques for inference control

5.1.1 Meta-level inference

Many problems in the field of Knowledge Based Systems, and Automated Reasoning stem from the generation of a combinatorial explosion in the search space. Problems arise if a tree representing *or* choices has many branches at each node. It seems clear that the sheer size of the search space may overwhelm the inference process. With a depth first strategy, if the "wrong" branches are picked at any point it may take an unacceptably long time to back up out of these dead ends before a solution is, eventually, found. In particular, if the search tree has infinite branches we may *never* find a solution. A breadth first search will also be explosive, in space and in time, if the solutions are deep down in the tree. Even an iterative deepening approach may be slow.

Configuration is a good example of this — the search tree usually has a high branching rate, with several choices in turn for processor, storage devices, termin-

als, and other devices; and very many choices later for how to connect up these devices. A poor decision in choosing the processor, say, may yield no solution (for that choice of processor) after an inordinate amount of time searching for one.

A simple inference engine may be built via declarative encoding of the factual information about a domain and nothing more. However, in many such cases we may be prone to the problem of combinatorial explosion mentioned above. When human experts tackle inference problems, they generally employ reasoning *about* the task and the domain to reduce search. Thus in many knowledge based systems it is often thought desirable also to encode meta-knowledge explicitly. One approach which has been tried for Prolog systems is to write a meta-level interpreter, rather than relying on Prolog's built in search strategy which may be inadequate in some circumstances. This interpreter incorporates meta-level knowledge constituting a suitable strategy for traversing the associated search space. For instance, a set of such interpreters was developed by the **Isambard** research group at Hewlett Packard Research Laboratories, Bristol. These were developed for use in protein topology (Owen, 1988).

Lowe (1988) examines the performance of these interpreters in a contrasting domain (electronic circuits). In both cases, domain specific control knowledge was encoded and used to control search *locally.*

5.1.2 Work on proof plans

Sometimes, however, more *global* control seems necessary, and the whole task or process needs to be planned before execution in order to avoid expensive search. The technique of *proof planning* has been used in mathematical domains, for equation solving (Silver, 1985) and inductive proofs (Bundy, 1988). Bundy *et al.* (1989) and Bundy *et al.* (1991b) give accounts of how these have been implemented.

Proof plans are a means of expressing the commonality between members of the same "family" of proofs, whilst allowing sufficient flexibility and adaptability to prove a large number of different theorems. They provide an expression

of strategies for automatic reasoning by describing *tactics* in terms of the *preconditions* under which they are *applicable*, and their *effects* if successfully applied. These specifications of tactics are called *methods*, and provide a basis for combining tactics (the effects of one implying the preconditions of subsequent methods) to form a complete plan which, if executed, will carry out a reasoning task, such as proving a theorem.

Tactics combine several low-level steps in a proof. For example, the tactic *wave/2* in the proof development system OYSTER (Horn, 1988) is given in Figure 5–1. We

```
wave(Pos,[Rule,Dir]) :-
        dequantify_all(Vars,Frees)
        then rewrite_at_pos(Pos,Rule,Dir)
        then requantify(Vars,Frees).
```

Figure 5–1: The *wave/2* tactic

see that it combines several small steps used in a proof. Tactics in configuration are analogous. We can have, for instance, a tactic to configure a device which performs several steps, most of which are not of interest above the engineering level — *e.g.* finding a suitable cable, *etc.* To the user, the tactic *configure_device/3* is simply that: one which configures a device in a configuration. The details of how, precisely, this is done are unimportant.

The *methods* of proof plans can be thought of as the specifications, expressed in a meta-logic, of tactics. At the planning level, we deal with methods rather than with tactics.

A proof plan embodies the expression of a strategy for automatic reasoning. Consider proof by induction. Although individual proofs differ, we can discern a common pattern. Figure 5–2 is a simple example of an inductive proof. Figure 5–3 is a schema for such a proof. Figure 5–4 shows the component "slots" of a proof plan method for induction. The *input* component indicates the form that the

If we have the definition of +:

0+x=x

s(x)+y=s(x+y)

and the wave rule

$$s(x) = s(y) \rightarrow x = y$$

Then *to prove:*

$\vdash \forall\, x, y, z.x + (y + z) = (x + y) + z.$:

use induction on x:

1. Base Case:

   $\vdash \forall y, z, 0 + (y + z) = (0 + y) + z$

   $\vdash \forall y, z, y + z = y + z$ (using base def. of +) (*base* in Figure 5–3)

   $\vdash$ *true* (*symbolic eval*)

2. Step Case:

$$\forall x\{\forall y, z.x + (y + z) = (x + y) + z \rightarrow$$
$$\forall y, z s(x) + (y + z) = (s(x) + y) + z\}$$

$x + (y + z) = (x + y) + z$ (induction hypothesis)

$\vdash s(x) + (y + z) = (s(x) + y) + z$ (induction conclusion)

$\vdash s(x + (y + z)) = s(x + y) + z$ (step def. of +) (*unfold*)

$\vdash s(x + (y + z)) = s((x + y) + z)$ (step def. of +) (*unfold*)

$\vdash x + (y + z) = (x + y) + z$ (wave rule) (*fertilization*)

$\vdash$ (*true*) (*fertilization*)

Figure 5–2: Proof of the associativity of plus

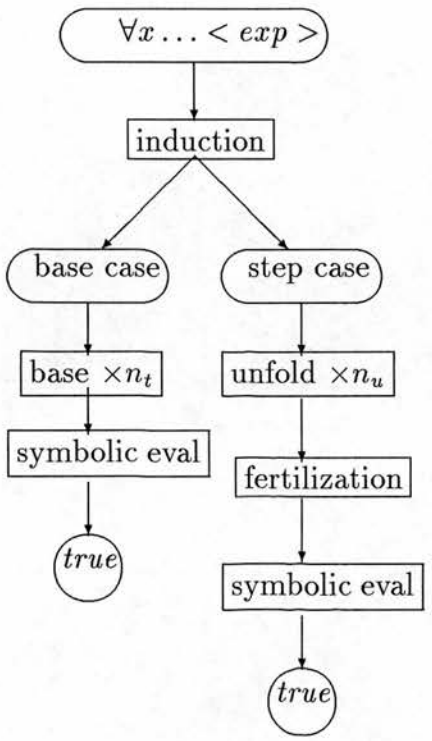

Figure 5–3: A general proof plan

Name	$induction(Scheme, Var)$
input	$\forall X formula$
output	$Base\text{-}form \wedge \forall X(Form \rightarrow Step\text{-}form)$
preconditions	*nil*
effects	$Base\text{-}form = replace\text{-}all(X, 0, Form)$ $\wedge Step\text{-}form = replace\text{-}all(X, s(X), Form)$
tactic	$induction(Scheme, Var)$

Figure 5–4: The Induction method

input formula must take for the method to be applicable, and the *preconditions* express any additional constraints. The *output* and *effects* describe the output formula in a corresponding way. The *tactic* is the name of the piece of procedure (which could be, for example, a Prolog program) which is applicable and results in the effects and output indicated.

Using methods written in a meta-logic facilitates the task of writing proof plans to guide the search for proofs in automatic theorem proving, ideally possessing the following properties:

1. Efficiency, because the combinatorial explosion is avoided, or at least greatly mitigated.
2. Generality, because a proof plan may be applicable to many proofs.
3. Maintainability, because the separation of factual knowledge from control knowledge means that either may be changed without affecting the other (*c.f.* the problems with XCON, below).
4. Explanatory power, because control decisions can be explained at the appropriate level, rather than by generating long chains of low-level choice points in the inference process.

These properties are important for *any* knowledge based system (Bundy, 1987). Thus proof plans can provide a useful vehicle for expressing strategies for problem-solving in other domains.

The process of identifying common structures and patterns of reasoning, encapsulating these, and testing them on a suitable corpus of theorems, is an incremental one. Adopting this basic philosophy of building methods essentially by *understanding* mathematical problem solving techniques, rather than by a succession of *ad hoc* heuristics and firepower (hardware), means that failures — to prove a given theorem, say — do not lead to dead ends in the process of developing a powerful theorem prover, but rather to a greater understanding and further enhancement of the theorem proving system: the extension of a method, the modification of preconditions, and so on.

Mathematicians, consciously or not, use heuristics to solve problems. They see that a theorem to be proved is similar to another one with whose proof they are

familiar, or they have an idea of whether applying a certain rule to an equation will make it easier or harder to solve from then on. Of course, there are many computer systems which incorporate heuristics, but they often do so in an inflexible way. The power of human mathematicians lies in their flexibility, and it is this that an AI theorem prover needs to capture.

I shall first explain the methodology of proof planning techniques. Consider a typical *induction proof*. Each proof consists of *base cases* and *step cases*; the elaboration of each of these is typified by the use of certain kinds of rewrite rules. This basic structure holds for a very large number of different proofs. The type of induction may be different (*e.g.* one-step, two-step, course of values) and the number of applications of the various rewrite rules may be different, from one proof to another; yet the proofs are all recognizable as belonging to one family.

A *proof plan* represents this generality and hence guides the proof of a theorem. In a typical theorem-proving system, there are so many inference rules that, without such guidance, there is combinatorial explosion in the search space. In addition, many of the rules may not be of much interest to the user of the system. In OYSTER, for instance, only some of the rule applications are "significant" proof steps, the remainder being trivial well-typedness checks. *Tactics* have been developed, and these assist the user in guiding the proof, taking care of many of the low level steps which are tedious and hard to keep track of.

Another example of a method (the *wave/2* method) is shown in Figure 5–5. This is the method specifying the *wave/2* tactic shown in Figure 5–1. The wave rule $Exp \Rightarrow NewExp$ referred to in the preconditions slot is a rule containing a "wave front" (put simply, this marks the parts of the the induction conclusion which are different from the induction hypothesis): for example in the example we gave in Figure 5–2 the available wave rules, with boxes marking the wave annotations (but see Bundy *et al.* (1991a) for details) are:

$$\boxed{s(\underline{x})} + y = \boxed{s(\underline{x + y})}$$

$$\boxed{s(\underline{x})} = \boxed{s(\underline{y})} \rightarrow x = y$$

Name	*wave/2*
input	$H \vdash G$ *i.e.* a sequent: H is the hypothesis list G is the goal
output	$[H \vdash NewG]$ a list because, in general, there is more than one output goal
preconditions	goal G contains a subexpression Exp to which a wave rule: $Exp \Rightarrow NewExp$ applies
effects	Exp in G is rewritten to $NewExp$ in $NewG$
tactic	$wave(Pos, Rule)$

Figure 5–5: The *wave/2* method

These rules attack the parts of the induction conclusion which differ from the induction hypothesis (called "waves") by moving the "wave fronts" — in the first rule the wave front is moved out; in the second it disappears altogether.

The CLAM system (van Harmelen, 1989) fits on top of the OYSTER proof development system (Horn, 1988) for intuitionistic type theory. OYSTER is a Prolog re-implementation of the Nuprl system (Constable *et al*, 1986) based on Martin-Löf (1979).

CLAM automatically puts together a sequence of methods for an input theorem in the form of a sequent $H \vdash G$, where H is a list of hypotheses and G is the goal, to form a *plan* for proving the theorem. If this plan is applied, the tactic associated with each method rewrites the sequent, in the same way as a user would apply a

refinement step when proving the theorem by hand, until the process terminates with complete subproofs at all the leaves.

The major achievement of the CLAM system has been that its flexible use of heuristics have enabled it to deal with many different theorems totally automatically, using relatively few, but powerful and general, methods, in a principled, non-*ad hoc* way.

5.1.3 Proofs of existence theorems

Of current interest in the development of CLAM is the technique of middle-out reasoning, since this is a useful technique in the proof of theorems of the form

$$\forall x.\exists y.p(x,y)$$

which has applications in program synthesis. An account of this work can be found in Hesketh (1991) and Bundy *et al.* (1990a). The contribution of *middle-out* reasoning is as follows:

In proving the conjecture

$$\forall input.\exists output.spec(input,output)$$

the usual method is to eliminate the quantifiers ($\forall$,$\exists$) to arrive at

$$spec(input,alg(input))$$

with *alg* as the *witness* for *output*. But how do we arrive at this witness?

The answer is to postpone deciding the first steps of the proof (*i.e.* deciding the induction schema) and concentrate on the middle part of the proof (hence the name "middle-out" reasoning). We use second-order meta-level variables to stand for the schema which become instantiated (using higher-order unification) in the course of carrying out these middle steps.

Work on synthesizing *logic programs* in WHELK, which synthesizes logic programs *via* inductive theorem proving, is described in Wiggins (1992) and Bundy *et al.* (1990b).

In synthesizing a configuration, we again have an existentially quantified variable: *viz* the configuration c, which becomes instantiated in the course of the proof. We should like to be able to generalize to the extent that work can be done on part of the proof without, necessarily, determining the structure of such a term (which depends on the processor, as we have seen, in Chapter 3, Section 3.3.2).

5.2 Proof plans in configuration

I have already shown (Chapter 4.3) that there is an analogy between one application of theorem proving (namely program synthesis) and configuration synthesis. The flexibility with which a proof planning system is able to use heuristics to guide the search for a solution will also be very useful in the configuration task. Let us take the analogy with the mathematician's thought processes described in Section 5.1.2. The expert configurer remembers examples similar to the current problem, and follows broadly the same pattern: she thinks in terms of a high level strategy (*e.g.* "now think about which disk drives are needed") to be followed at a later stage by more detailed considerations (*e.g.* "do I need a cable for this disk?"), and uses heuristics to guide the search for an acceptable configuration.

In configuration problems, as with theorem proving, we find "families" of problems, with a common structure overlaying the variability of the individual steps in different problems within the same family. In this work, the additional constraint, that of coming up with "good" solutions, according to some criteria, was very important. For instance, we might *prefer* configurations which were *efficient*, or which *cost less than* $£X$. It was found necessary to develop proof plans to deal with these different scenarios. These kinds of consideration were not prominent in early proof plan work on theorem proving, although they are a powerful motivation in transformation (Madden, 1991). Here, a simple proof from which a algorithm may be extracted is transformed to a more complex proof: for instance, by using a different form of induction. The complex proof may well give a more efficient algorithm. Finding the simplest proof and then transforming it to one

giving a better program is one of two possible approaches, the other being actively to seek a proof leading to a "good" algorithm from the start. It is possible that the work done in tackling this problem in the configuration domain reported here may inform further research in other domains, including mathematics.

5.2.1 Tactics

Tactics for configuration are analogous to tactics in mathematics, in that they are programs which combine a significant proof step with other low level steps which are of less interest, and probably harder for the user to keep track of (see Section 5.1.2).

This is best illustrated in an example. Let us look at the tactic *configure_device/3* given in Figure 5–6.

```
configure_device(Device,IC,C):-
        connect_cable(Device,Cable,C),
        connected_via(Device,IC,C),
        type([Device,Cable,IC],_).
```

Figure 5–6: The tactic *configure_device/3*

The three arguments of *configure_device/3* are, in order:

1. a device to be configured, *Device*,
2. an interface channel, *IC*,
3. the configuration in which the device and its interface occur, *C*: *i.e.* the configuration that we are synthesizing.

This tactic can be used in any of the following modes:

- In the case where *Device* and IC are ground terms; to check whether a device *Device* is *configured* in the configuration C via an interface IC.

- When *Device* is ground and IC, initially unknown, is ground in some subterm of C; to find out which interface is employed in connecting *Device* in C.

- When *Device* is ground and IC, initially unknown, may or may not be ground in some subterm of C; to search for a suitable interface IC in order to connect up *Device*.

Whichever the mode, these are the conditions for a device to be, or to become, connected in a configuration C in every case:

1. It must have a cable.

2. It must be connected via its cable to an (the) interface.

3. This connection must be well-formed.

Condition 1 is checked (in the case where the cable is ground) or ensured (in the case where the cable is uninstantiated) by the *connect_cable/3* goal in Figure 5–6. The argument *Cable* refers to the cable, or other means of connection[1], of the device.

Condition 2 (the *connected_via/3* goal) checks that there is a connection, and condition 3 (the *type/2* goal) checks that this connection is *well-formed*, in the sense defined in Chapter 4.6.3.

These conditions are, to the average user, trivial and uninteresting: to the average configuration engineer they are of the utmost importance, but may be hard to keep track of and easy to get wrong.

---

[1]In this sense, *cable* is an abstraction. It could be, for instance, a modem connection, involving more complex electronics.

Other tactics include those for configuring processors, memory, interfaces, and other components; finding devices to meet specifications; *etc.* In each case, there are several steps involved, of which many may be trivial in the sense explained above.

5.2.2 Methods

A method is a specification for a tactic. Figure 5–7 gives a method for the tactic we have already seen in Figure 5–6. We see that the goal to be proved must have

Name	*configure_device*
input	$configure(Device, IC, C)$
output	*nil*
preconditions	*Device is a device of* C *and* $IC : \tau$ *and Device needs slot of type* τ *and the number of slots available of type* τ *is* $s(n)$
effects	*the number of slots of type* τ *available is* n
tactic	$configure\_device(Device, IC, C)$

Figure 5–7: The method *configure_device/3*

the form *configure(Device,IC,C)* and that the output of the method is *nil*, which means that this is a terminating method, since if the device is configured there is nothing left to do. The most important precondition is the last: the number of vacant slots is $s(n)$, *i.e.* the successor of some natural number n. Since $s(n) > 0$ this means that there is at least one vacant slot of the correct type. The effect of the method is to reduce the number of slots by one (from $s(n)$ to n).

Chapter 6 compares the inference process carried out by the method with that carried out with the corresponding tactic. We see there that the latter has many

more steps, thus there is a saving in time and effort at the planning level when using methods rather than tactics.

5.2.3 Configuration plans

If we take the simple example we have already seen in Chapter 4.9, *viz.*

$$spec(C) \equiv (processor(C) = example\_proc(N)) \wedge (capacity(disks(C)) \geq 500) \tag{5.1}$$

the plan formed will be as follows.

1. Configure a processor and set up a "skeleton" configuration.
2. Find disk drive x which has capacity at least 500Mb.
3. Configure x somewhere (details, such as the cable needed, or the socket used, are not given at this, the planning, stage).
4. Select a system disk y.
5. Check that y can be configured somewhere.
6. Select a tape z for backup.
7. Check that z can be configured somewhere.
8. Check that we now have a legal configuration.

This plan consists of the names of all the methods, with some arguments instantiated (for instance, the names x, y, and z) and others not (for instance, the names of the interfaces which connect them). If we were to execute the plan, we get the complete configuration, with all the details filled in — including the names of the interfaces, and the cables used.

5.3 Semantics

The separation of planning and execution represents the two separate views of the problem:

1. The planning stage corresponds closely with the customer, or user, view;

2. The execution stage corresponds closely with the engineer view.

Although the planning stage does not give enough detail to configure the hardware, it contains sufficient information to allow the user to decide whether the plan, if executed, is likely to yield an acceptable configuration. For instance, the user can see which devices are being provided, even if the details of how they are connected are absent. In essence, the ability to plan weeds out most, if not all, of those solutions which would be unacceptable to the user, without expending a large amount of time executing the plans, *i.e.* actually configuring such systems. Suppose the first solution at the planning stage gave us at least one device we did not want. Then we would seek further solutions. Or we might want to see what the options were. Whatever our motivations, we would not want to wait while all the devices were connected up by the configurer before we saw what devices had been chosen. We are not at all interested in the configuration details: rather, we want to look at the "bare bones" of each solution before deciding whether this is the one we want or else passing on. With, say, a choice of ten disk drives and three tape drives this gives thirty different combinations which is quite enough.

Since it is much faster to plan than to execute (see Appendix F), this planning stage is a kind of "rapid prototyping" leading to a likely looking plan, which can then be executed.

To recap, we can *roughly* see the two stages as

1. From high level user specifications, deciding which devices will provide these, *i.e.* how we get the required usage, and, in addition, some handle on how we

provide the required utility (*e.g.* knowing whether devices must be connected according to efficiency heuristics, or not).

2. From knowledge of which devices appear in the configuration, and a few constraints on how they may be connected (*viz.* the utility considerations mentioned in 1.), deciding how they will be connected.

5.4 Separation of knowledge and control

Recall (Chapter 3, Section 3.9) that the knowledge used to carry out the task of synthesizing computer configurations is of three kinds:

1. Object level axioms, *i.e.* hard and fast rules, such as "all devices must be connected"; "there must be no more than four disk drives connected to this kind of channel";

2. Heuristic knowledge, *e.g* "for an efficiently running configuration, do not connect more than three disk drives to this kind of channel";

3. Control knowledge, *e.g.* knowledge about the order in which subtasks should be carried out, such as "synthesize devices before card cages";

and that it is important that these three categories of knowledge be kept separate and explicit so that they may be easily maintainable. (See, for example, Chapter 7, Section 7.3.3.)

It is clear that object level knowledge must be updated as new products come into being and others become obsolete. Chapter 4 described how this changing knowledge is represented and Chapter 6 will show how it can be updated.

Heuristic knowledge is, or should be, changing with time and circumstance; for instance, the fact that particular configurations lead to inefficiency may only be learned from experience of actual running configurations.

Explicitly and separately held control knowledge enables us to update the configuration as whole structures or new kinds of products are added or altered, and may mean that the system can be generalized and used for other, similar tasks.

I turn now to the question of how the last two of these should be represented and used in synthesis.

5.4.1 Heuristic knowledge

We saw, in Section 5.4 points (1) and (2) that there may be both "hard" and "soft" limits on, for example, how many disk drives can be attached to a particular kind of channel. Four is the actual "hard" limit — any more and the configuration would not be legal — but experience has shown that configurations configured up to the legal limits run slowly. On the other hand, configurations with three disk drives connected per channel do run efficiently. We might have, therefore, a "soft" limit of three disk drives per channel.

As a general strategy, we can see that, all things being equal, a configuration obeying the soft limits is preferable and it is that kind of configuration we should go for first. However, if this is not possible, due, for example to cost constraints, or to sheer size of configuration, one or more soft limits will have to be ignored. The management of soft constraint relaxation is not trivial, as I pointed out in Chapter 3. There are many design issues to be considered.

Our solution to this problem, as discussed in Chapters 6 and 7 is to have alternative methods: one applying only soft constraints, the other allowing relaxation of these, but obviously still applying hard constraints. This gives us methods which configure components according to efficiency heuristics as well as methods which simply ensure legal configuration. This necessitates a strong control strategy to ensure that the methods are tried in the right context and in the right order. The use of supermethods, which are methods which specify the use of other methods, deals with this issue.

5.4.2 Control knowledge

Even in cases where we *can* design a configuration obeying all the soft constraints, the solution space may still be large. We need to decide which additional factors to look at in deciding a final solution, or small set of alternatives. We need to incorporate the knowledge needed to support the designer of computer configurations in her decision making.

Much guidance in searching for a configuration can be provided if the user of the configuration system has facilities for scene setting; *e.g.* stating whether an efficient configuration is important, whether there is a cost constraint, *etc.* There is a small subclass of design problems in which *the* "optimum solution" (along one of several possible different dimensions: cost, efficiency,...) which meets the specifications is required. The underlying strategy to be used in this design scenario is quite different from, say, that to be used if the problem is to find any satisfactory solution(s) subject to keeping within a cost ceiling.

In what follows, I have considered only cost and efficiency as design issues in order to simplify the discussion. There are other factors, not mentioned here; for example:

- The *expandability*, or potential growth path, of a configuration, *i.e.* its capacity to be augmented at a future date.
- Technological issues: preferring innovative products , or avoiding potentially obsolescent components.

5.5 Strategies

5.5.1 Constraint relaxation

I give an example which demonstrates the need for strategies to relax soft constraints when this proves necessary, whilst avoiding sub-optimal solutions.

Let us look at two soft constraints:

1. Whereas four disk drives per HPIB channel are legal, more than three can, in certain circumstances, give an inefficient configuration.

2. For a two card cage configuration, more than six interface cards can, in certain circumstances, give an inefficient configuration.

These soft limits are more stringent than the "hard" limits. They are used by heuristic strategies aimed to give "better" configurations, usually in the interests of efficiently running configurations. If possible, we configure systems so that the soft limits are adhered to. However, this is not always possible, and we have to relax one or more of these soft limits. In the next section I examine the management of this constraint relaxation, with regard to certain pitfalls which must be avoided.

5.5.2 An example

In this example, I shall focus on the part of the configuration task which connects disks to the configuration.

Each disk (assuming that it has a suitable cable) may be connected to any existing HPIB channel, provided there is room, *i.e.* the limits for that channel will not be exceeded. We would prefer that the soft limits are not exceeded, but must not exceed the hard limits in any case. Alternatively, we can try to generate a new HPIB channel and connect the disk to that. To do this, there must be space on an existing card cage (or we can again generate a new card cage; for the sake of brevity we shall assume that this option is not open to us — true examples exist in which the number of card cages is already at a maximum, or where cost constraints preclude the addition of another card cage).

Suppose the situation so far is as shown in Figure 5–8.

Here, there are two card cages, with various interface cards inserted in slots: one card cage has two MUXes, one LANIC, and one HPIB (these terms are explained in Appendix B.1); the other has two HPIBs configured. Each HPIB channel has three disk drives connected, and two of them have a tape drive connected.

We observe that the following hard constraints are satisfied:

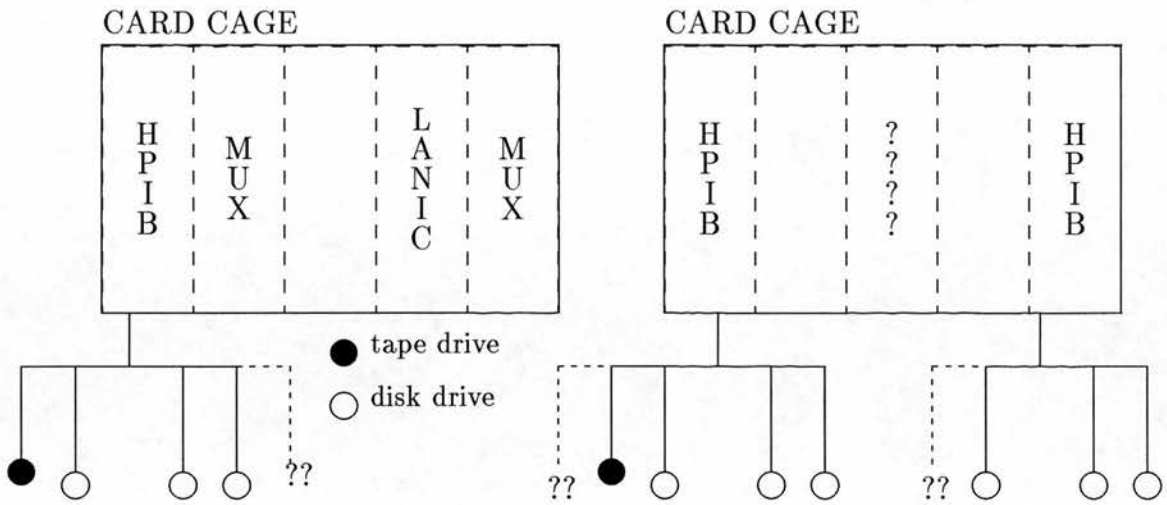

Figure 5–8: Situation before the next disk is added

- No more than ten ICs in total.
- No more than four HPIBs per card cage.
- No more than six devices per HPIB.
- No more than four disk drives per HPIB.

Moreover, the following soft constraints are also satisfied:

1. No more than six ICs in total.
2. No more than three HPIBs per card cage.
3. No more than five devices per HPIB.
4. No more than three disk drives per HPIB.

Suppose now that we wish to add a further disk drive to this partially configured system. Soft limit 1 will be violated if we add another HPIB channel. Soft limit 4 will be violated if we add another disk drive to any of the three existing HPIB channels. It is therefore necessary to break one of these soft limits in order to configure the next disk.

We consider two solutions.

1. Overload the third HPIB channel.

2. Overload the card cages by adding another HPIB channel.

Which of these do we prefer? The answer is contingent on circumstances. In general, both are semi-acceptable as solutions. It would take an expert on performance to decide which to prefer and I shall not go into the detail here as it is dealt with in Chapter 3, Section 3.6.4.

Let us carry on, and assume that there is one further disk drive to connect. Assuming that the situation is as in solution 1, we have the same choices as before. If we overload another channel, then we have violated one constraint, twice, involving two separate channels. (In this example there is not the option to violate it twice on the same channel because of hard constraints.)

If we choose to violate constraint 1 this time, however, then we arrive at the solution shown in Figure 5–9.

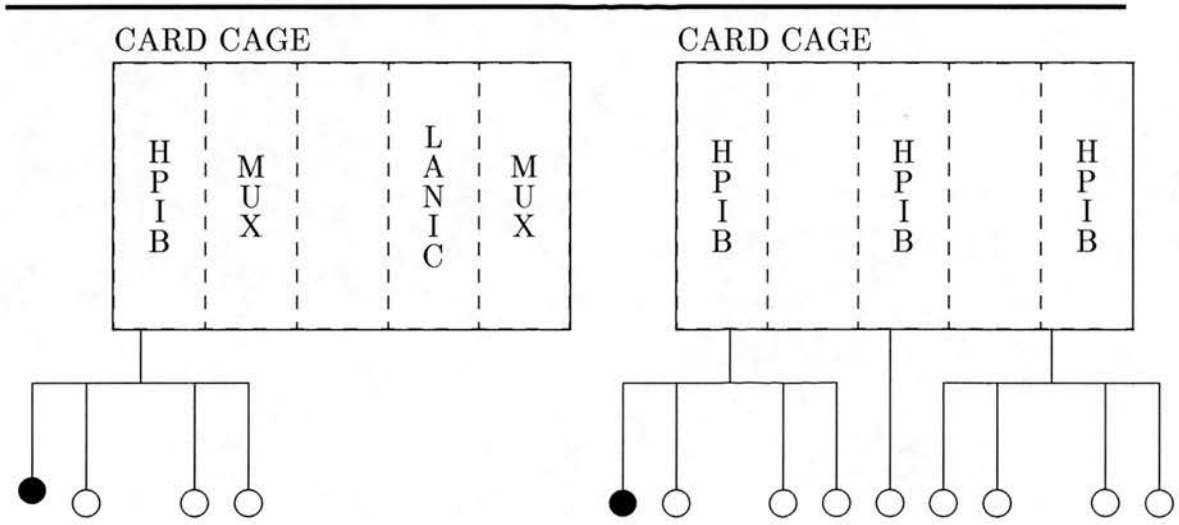

Figure 5–9: Locally sub-optimal solution

However, this solution is locally sub-optimal, in the sense that if we moved one of the disk drives from the overloaded channel to the newly configured one, we would have a solution with the same components (and same cost) which was better: only one constraint being violated, rather than that same constraint plus another one.

Note that if we chose the other path, namely Solution 2, no further constraints need be relaxed when configuring the last disk drive, since it can be placed on the new HPIB channel without violating any further soft constraints.

These points emerge:

- One solution, which is locally optimal, violates constraint 4 twice.
- Another, again locally optimal, solution violates constraint 1 once.
- Whereas both the above are feasible as solutions, we must avoid the situation of Figure 5–9.

We need to bear these points in mind when drawing up methods; in particular when considering "supermethods", or proof plans, which embody planning strategies.

5.5.3 Proof plans to express strategies

In tackling the problem of directing the search towards solutions which are, at the very least, locally optimal, I need *strategies*, which are expressed in terms of proof plans, which can be thought of as comprising "supermethods". A supermethod has a structure as outlined in Figure 5–10. The main feature is that the *effects* call other methods. Now, without this supermethod, the planner is capable of finding a plan which consists of the sequence

$$method_1, method_2, \ldots, method_n$$

in place of the supermethod *basic_config/1*. Hence *basic_config/1* can be thought of as *equivalent* to the proof plan so formed. I have embodied a few strategies as supermethods. For example, there is a supermethod for configuring all system devices in accordance with heuristic guidelines. There are other supermethods which will, if necessary, relax one or more soft limits whilst still keeping within the legal limits. I found that reasoning under cost constraints requires different strategies according to whether there is a given upper ceiling or whether it is required to minimize cost. The planner will build, from the methods available, a

Name	*basic_config*
input	$\exists c.spec(c)$
output	...
preconditions	...
effects	... $applicable(method_1)$ $applicable(method_2)$ ... $applicable(method_n)$ ...
tactic	*basic_config*

Figure 5–10: Structure of a supermethod

customized plan appropriate to the particular specification given. For example, if the top-level proof plan, which turns out to have a wide area of applicability, is:

configure-processor/2

then *configure-template/1*
then *configure-terminals/1*
then *configure-disks/1*
then *configure-tapes/1*
then *configure-printers/1*

then this can be encapsulated as the supermethod which reflects the strategy of

1. Deciding what an appropriate processor would be from information given in the specification.

2. Each processor fixes a kind of "template" configuration, *e.g.* a 950 processor will support certain types and models of devices, will have at least two (and at most four) card cages, *etc.*

3. Attend to user needs explicitly given in the specification, configuring devices as explained in Section 5.2.1, adding in essential components, *e.g.* memory, back-up devices, not given explicitly.

Within the overall guidance given by this plan, there is flexibility: *e.g. configure-tapes/1* will use the methods appropriate for the specification given; either configuring tape drives explicitly given by the user or inferring a suitable tape configuration for the disk configuration synthesized if the user has not explicitly specified them.

The existence of supermethods reduces the chances of a random application of methods which could lead to locally sub-optimal solutions; in this sense the supermethods embody strategies for configuration. However, if the supermethods are not applicable, it is possible to slot together methods which will synthesize *some* configuration.

Chapter 6, Section 6.8.2 gives several examples of strategies which were found useful and which have been captured as proof plans.

5.6 Summary

I have implemented the object level theory of configuration in a way which ensures legal configurations whilst facilitating an architecture for encapsulating knowledge for designing computer configurations. I have identified some of the key design issues in building a system to support the synthesis of well-designed computer configurations. Although I aimed to automate the task in the first instance, I nevertheless developed this methodology with an eye to enabling future systems to be capable of allowing co-operation between designer and machine. The aim of my current system is complete automation, whereby the user may ask for as many different solutions as she likes, the ideal being that acceptable solutions will be presented early on, amongst the first few solutions given by the system. The acceptable plan can then be executed. However, in a co-operative system, some

of the choice points could be removed *in situ* by allowing the user to refine the specification interactively.

It is very difficult to maintain large knowledge bases for configuration tasks, unless there is a clean separation of object-level facts from meta-level control and heuristic knowledge. In Chapters 4 and 5 I have outlined how both levels of knowledge may be extracted and formally represented for the task of synthesizing computer configurations to meet specifications. Utilizing a technique analogous to program synthesis, whereby the desired configuration is synthesized as a byproduct of the proof of the specification theorem $\exists c$:**configuration**.$spec(c)$, where c is a term of type **configuration**, I can incorporate meta-level techniques for theorem proving. Thus I reduce the problem of combinatorial explosion which exists for object-level search, whilst tackling some of the design issues of the domain. Such an approach makes the task of maintaining knowledge based systems more tractable and reliable. Heuristics and knowledge about configuration strategies are incorporated into this framework in a principled way to generate a useful subset of solutions out of a potentially very large set of possible solutions.

Chapter 6

Implementing a configuration system

Experience is the name every one gives to their mistakes.

Oscar Wilde

6.1 Introduction

In this chapter I describe the implementation of the CLEM system, which configures a subset of HP3000 systems. My implementation was carried out in several stages. The result to date is a prototype system, in Prolog, which can handle a variety of specifications, and which can employ strategies for control and for taking design issues into account. I tested the maintainability of the system by adding a new processor and other components. This tested whether the object-level knowledge could be updated independently of the rest of the system. Maintainability of control information was tested throughout: control information tends to be learned gradually, and I was able to incorporate such understanding on an incremental basis, without needing to make major changes to the rest of the system.

In this chapter, I shall refer frequently to Appendix C. This is a slightly modified (to remove redundant explanatory material) version of Lowe (1993), the manual for programmers and users of CLEM.

The following lists the stages of development of CLEM:

1. Study of two contrasting HP systems.
2. Encoding the object-level theory.
3. Testing the object-level theory on very simple specifications and very small systems.
4. Writing tactics.
5. Specifying tactics by methods.
6. Encoding heuristics: incorporating *soft limits.*
7. Testing the planner for simple specifications with no global goals.
8. Evaluating the comparative efficiency of the planning and execution stages.
9. Evaluation of the quality of design of the solutions obtained by planning over blind object-level search.
10. Evaluation of the efficiency of search at the planning level.
11. Encapsulating proof plans for basic strategies as supermethods.
12. Evaluation of the quality of design of the solutions obtained using supermethods.
13. Evaluation of the efficiency of search using supermethods.
14. Adding strategies to deal with global goals such as cost and efficiency.

In this chapter, I give simply an account of what appears in the current version of CLEM, leaving evaluation and discussion of various alternatives to Chapter 7.

6.2 Study of two contrasting HP systems

6.2.1 Choice of test data

I studied the configuration of two different systems, the HP Series 70 and the HP3000 Series 950. The former is an older system, and is characterized by having more constraints in the placing of interface cards in card cages; for example, the order of cards is important. The 950 is a much newer system and there are fewer such constraints. The search space involved with both can be quite large, as they are both designed as big multi-user systems. I decided that either would provide a good test for my hypothesis that that meta-level reasoning techniques were effective in dealing with the problem of combinatorial explosion, since search at the object-level was likely to prove difficult. I had ready access to the domain

information needed for the 950, whereas this was harder to obtain for the Series 70 as there was less interest in it from researchers at HP. In particular, the knowledge necessary to incorporate strategies for incorporating meta-level knowledge was much more readily available for the newer systems. This is necessary to "fine-tune" the system to make it efficient.

However, in order to ensure that I had access to the information I needed, *i.e.* knowledge about configuration strategy as well as details of components, it was important that I chose a system where I could guarantee such access. For logistic reasons, therefore, I decided to use the HP 3000 systems for my study, where there was a fair amount of expertise on tap. This also had the advantage that I would be concentrating on the more up-to-date concerns of design, rather than the older ones of engineering detail — the exact placement of interface cards, and so forth. Nevertheless, it was useful to have studied the Series 70, from the point of view of developing as general a system as I could; it made me aware of the variations between different hardware configurations based on different processors. This turned out to be important for maintenance.

6.2.2 HP 3000 hardware systems

The 900 range comprises hardware systems based on various processors, of which the 950 is currently the second most powerful. It supports up to 400 user ports, although the number of active users (*i.e.* logged on and using the system) is normally much less than this — about 200. As a rule of thumb, the number of active users can be taken to be around half the number of ports, bearing in mind that users can be logged on yet inactive. Another rule of thumb is that a 950 processor is suitable provided there are at least 100 user ports required. My interpretation of all this is that the 950 is not really suitable if there are *less* than 100 users. However, since this is only a heuristic and the object-level theory allows it, in the early stages of development many test examples were run in which only a small number of users were specified.

Note that the object-level theory does not *directly* supply us with an absolute upper limit on the number of users for the 950. This is for two reasons:

1. Ports, in general, may support *either* terminals (users) *or* printers and other output devices. How many of the former we are allowed depends on how many of the latter we need.

2. The total number of ports will not, in general, be static over the lifetime of the system, and in some systems (not the 950) the interfaces for ports compete for resources — by which usually we mean card cage slots — with other parts of the configuration; for example, with memory boards.

It can be seen that the choice of processor is critical and it is important that our implemented system is efficient in making use of information early on which directs us towards a sensible choice.

6.3 Encoding the object-level theory

6.3.1 Component set

I first put together files containing factual, object-level information. I had decided upon a subset of the HP3000 range, namely the 950 processor and all components which can form part of a 950 configuration. To this I planned to add the 925 processor and any additional components associated with the 925 which I had not already incorporated. I ascertained that the 925 manuals and component lists were available, but looked no further at them until after I had developed the system for the smaller subset. This would test one aspect of the maintainability of the system.

6.3.2 Types of components

I had, then, a list of components. The first important thing to record about each was its type. Appendix C.3.1 describes the implementation of types (as proposed in Chapter 4, Sections 4.5 and 4.6). in CLEM, and gives some examples of types and their usage.

The first section of the file `types.pl` records the type of each simple object in the theory; every such object is a member of exactly one type. Examination of the code in Appendix D.1.1 will reveal that some types are fairly sparsely inhabited. This is a result of the extremely flat typing I decided to adopt, but may also reflect the newness of the computer systems in question.

To summarize, we have:

- the **processor** and **memory** types;
- device types **disk**, **tape**, **printer**, **terminal**;
- various interface types —
  - **mux** (in some systems, for instance in the 800 series, these are used to connect up terminals; in the 950 they are only used for the console and a printer for use by the computer operator);
  - **lanic** (used in the 950 for terminals and serial printers);
  - **fchannel** (for certain devices which may be connected *via* a fibre-optic link); and
  - **channel** (for system devices not so connected — note that only one member of this type, the HPIB, is recorded here, but older systems, such as the Series 70 mentioned in Chapter 3, use GICs);
- connectors:
  - **serial_conn** (for insertion in a **lanic** component, and in which port-group connections are in turn inserted); at present I only know of one member of this type, the DTC;
  - **rs_232_portgroup** and **rs_422_portgroup**. These are for connecting cables and modems (but for an explanation of how modems fit into this picture, see Chapter 7, Section 7.3.2);
  - **cardcage**: each of the members of this type is peculiar to one given processor;
  - cables: **rs_232_cable**, and **rs_422_cable**. There are vast numbers of members of these types. Note that a modem is a member of the former type (see again Chapter 7, Section 7.3.2).

In order to generate particular instances of types of components from functions, *e.g.*

$$7937H : \mathcal{N} \rightarrow \textbf{disk}$$

I used the routine shown in Appendix C.8.1 to generate a natural number which would give a new instance of that model of component.

Next we need to identify union types. It has been explained (Chapter 4, Section 4.6.2) that these are used in defining certain list types. We have identified the union types

- **sdisk** ≡ **disk** ∪ **fdisk**: in the manual, these are identified as disk drives connected *via* system cables and fibre-optic disks respectively.

- **device** ≡ **sdisk** ∪ **tape** ∪ **printer** ∪ **terminal**: all devices; except that I did not incorporate any output devices other than printers (I could have included plotters, *etc.*).

- **ic** ≡ **mux** ∪ **lanic** ∪ **channel** ∪ **fchannel**: these are the interface cards.

- **portgroup** ≡ **rs-232-portgroup** ∪ **rs-422-portgroup**: portgroups with modem ports as a member of the **rs-232-portgroup** type.

- **serial-cable** ≡ **rs-232-cable** ∪ **rs-422-cable**: cables (including modems).

The next kind of information, which is also to do with type, is how components may be connected. The information needed depends on the type, or the union type, of the object in question. For example, anything of type **disk** can only be connected *via* a cable of type **system-cable**. A terminal can only be connected *via* a cable of type **serial-cable**. A printer may be connected *via* a cable of type **system-cable** or of type **serial-cable**. This is expressed by cross-product types.

It is possible that there are exceptions. For example, maybe there is a printer which cannot be connected *via* a modem. Thus in x :**printer**$\times\tau$, the type τ depends on x. An example of this is given in lines 268–275 of Appendix D.1.1.

In turn, the cross-product terms formed in this way may be combined into lists and lists may be combined with interfaces. An example of this is the type **channel**$\times$list(**device**$\times$**cable**). See Appendix D.1.1 for examples.

6.3.3 Attributes

Some components have attributes which must be recorded. For example, disk drives have a given capacity; printers have speeds; and so on. About 70 attributes (capacity, printer types, linespeed, and so on) are currently recorded by CLEM.

For a component such as a channel, or a card cage, there are limits associated with them — for example, the limits of channel may be six of type **devices** and four of type **disk**; the limit of a card cage may be six of type **ic** and four interfaces of type **channel**. These attributes are stored by CLEM. Examples and explanatory material can be found in Appendix C.3.2.

The soft limits are recorded elsewhere, in a separate file. These are all soft limits on the number of components which may be connected together with other components, aimed at guaranteeing efficient configurations. This is described in Appendix C.4.

Note that a component may have more than one limit associated with it. Limits are functions of both the components which is limited and the *type* of the components connected to it. So, for example, we see

- limit($hpib(\_)$,**device**) = 6
- limit($hpib(\_)$,**disk**) = 4

6.3.4 Rules

There are various rules for connecting components together, and for accessing (projecting) components which form part of a configuration, including axioms for determining what constitutes a valid configuration. These are explained in C.3.5.

6.4 Testing the object-level theory

The first stage was to test the correctness of the object-level theory. I began by testing a very small system (the one given in the example of a synthesis proof in Chapter 4, Section 4.9). Other examples gradually increased the required disk storage capacity in the specification, and varied the type of disk drive between **disk** and **fdisk**. The full details of the test procedure, results, and analysis are in Chapter 7, and Appendices E and F. Here I only mention the problems which were found in using the object-level theory alone, in order to motivate and introduce the techniques used to overcome these difficulties.

As expected, the difficulty here lay with the combinatorial nature of the search space: this is one of the reasons why I claim that the explicit representation of meta-level reasoning is highly desirable in this domain using my approach. I therefore went on to develop meta-level support tools so that I could synthesize realistic examples.

6.5 Writing tactics

The implementation of tactics is explained in C.6.1. They follow the rationale already outlined in Chapter 5, Section 5.2.1.

6.5.1 Configuring processors

To begin with, I did not implement any rules of thumb for choosing the processor. It is worth pointing out here that the usual criterion is the number of terminals to be connected in the configuration; the 950 processor is indicated when this exceeds 100. However, there are cases when a 950 would be demanded for a lesser number of terminals: maybe extra processing power is needed for some reason, or maybe it is thought that the system will be expanded by adding more terminals at a future date. Hence the simple tactic

```
configure_processors(P,C):-
     groundp(P),
     processors(C,P)
```

was used, to cope with the case where the processor is given in the specification, as in

```
processor(C,P) ∧ ...
```

Later we added the tactic:

```
choose_processors(P,C):-
     no_users(C,N),
     N >= 100,
     processors(C,['950'(_)]).
```

— one clause for each processor in the knowledge base. Note that, if $N \in \mathcal{N}$ is the number of users of a configuration, then it is not true to say that the processors partition $\mathcal{N}$ into disjoint sets: there will be more than one choice of processor, in general, for each N. This is perfectly reasonable, and in fact is a useful feature of the system, as at the moment most configuration aids rely on the experience or intuition of the sales representative to decide which processor(s) may be suitable.

6.5.2 Configuring devices

The tactics for configuring devices consist of the following steps:

1. Find a "cable" (this is an abstraction for a connection path, and may be, literally, a cable, or something else, like a modem, or a fibre-optic link).
2. Find a "slot" (socket) for the device-cable combination.
3. Check that the resulting object is well-formed, *i.e.* in a type.

There may be several alternative ways of configuring a device. For example, a printer may be connected via an object of type **system-cable** to an object of

type **channel**, or else via an object of supertype **serial-cable** (which may be of type **rs-232-cable** or **rs-422-cable** to an object of supertype **portgroup** (of type **rs-232-portgroup** or of type **rs-422-portgroup**). It is possible for the specification to be drawn up so as to exclude some of these possibilities, *e.g.* by insisting that a system-cable be used for the printer.

6.5.3 Configuring connectors

The tactics for configuring connectors, such as interface cards, portgroups, *etc.* are similar. The component is added, if necessary (there is a check that it is not already part of the configuration), and a "slot" is found for it, in a card cage, or DTC, or LANIC, depending on what it is, so as to synthesize a well-formed object in the configuration.

6.5.4 Configuring devices to meet specifications

The tactic `match_attributes/2` is designed to locate a device with a particular attribute. For example, the specification may include the goals

$$(printers(C) = [P_1, P_2]) \wedge (lpm(P_1) \geq 600) \wedge (cps(P_2) \geq 480)$$

i.e. there are two printers in the configuration, of which one has speed of (at least) 600 lines per minute, and the other has speed of (at least) 480 characters per second.

Other tactics of this kind are terminals specified by application; printers specified by application; terminals specified by characteristics such as offering graphics facilities; printers offering facilities such as double-sided printing, *etc.*

This tactic

```
match_attributes(Goal,Device)
```

works by matching `Goal` (for example `lpm(Device,Speed)`) against one in the knowledge base, thus instantiating `Device`. There are examples of its use in Appendix C–9.

6.5.5 Configuring devices to meet overall specifications

The tactics `disk_storage/3`, `total_disk_storage/4`, and `tape_capacity/3` ensure that the configuration includes devices which give the required utility, *e.g.* disk storage of the capacity given in the specification.

At the object-level, we have only generate-and-test as a means of synthesizing devices to meet a certain requirement: *e.g.* they synthesize a list of devices and compute their total capacity. However, the tactics use more efficient means — for example, choosing a model of disk and generating a list of such disks to the required capacity. As well as being more efficient and less random, this gives better designed solutions: it is not considered good design to mix disk drives in a configuration. Note that the user can still specify mixed disk drives explicitly.

Of course, using tactics in this fashion we lose completeness, but in a wholly desirable way. We prune the search tree in a way similar to the human designer of hardware systems. For example, there are an infinite number of ways of satisfying the specification

$$disk - capacity(c) \geq 1000$$

but a sales representative given such a specification with no indication as to what models of disk drive to use would *never* consider solutions including a mixture of disk drive models. Thus it seems sensible for the corresponding tactic only to generate lists in which all elements are instances of the *same* model of disk drive.

6.5.6 Configuring explicitly named devices

This tactic works on goals of the form

$\langle device\text{-}name \rangle(c) = \langle device\text{-}list \rangle$,
e.g. $disks(c) = [7937(\_), 7936(\_)]$,

checking that the devices named in *device-list* are of the correct type for the functor *device-name*. The early configuration systems, which are no more than aids, take their input exclusively in this way, although they do prompt the user with acceptable alternatives.

6.5.7 Other tactics

The configuration may be "under-specified" in that various essential components are not mentioned. The tactic `is_legal/1` checks that all such components are present. This may fail sometimes, unless the tactics `add_system_disks/2` and `add_backup/2` are employed first. Note that these are different from the tactics `system_disks/2` and `tapes/2`, which are used in conjunction with explicit user specifications. The difference here is that `add_system_disks/2` and `add_backup/2` use heuristics to decide on *sensible* additions; the user, on the other hand, is free to specify *any* legal values for these.

6.6 The meta-level

6.6.1 Specifying tactics by methods

Methods were drawn up which specified each tactic, along the lines indicated in Chapter 5, and these are given in Appendix D.2.2. In this chapter we will give more detail than is to be found there about the preconditions for the methods and the rationale behind them.

6.6.2 Method for configuring the processor

The preconditions for `configure_processor(Proc,Config)` require that we have a goal of the form `processors(Config,proc)`, with `proc` ground. Otherwise a method `choose_processor/2` may be applicable.

In fact, the only case in which it was not applicable would be if there were no users of the system (as measured by the number of terminals, which can be deduced from the specification) so one of these two methods will always be applicable in practice.

Name	{h_}configure(_serial)_device(Device,Interface,Config)
Input	configure(Device,Interface,Config)
Preconds	check not configured/configurable by alterative means
	slots of right type available
Effects	reduce slots by 1
Output	nil
Tactic	{h_}configure_serial_device(Device,Interface,Config)

Figure 6–1: Methods for configuring devices

The config_processor/2 rewrites the goal to template(Proc,Config) at which point (and only then) the method special(Config) is applicable, which instantiates Config to a 'skeleton' configuration.

It is important for the sake of efficiency that the configure_processor/2 method precedes the choose_processor/2 method otherwise user-preference may be ignored at first; we must assume (however wrongly) that users, in explicitly choosing a processor counter to heuristics, know what they are about.

6.6.3 Configuring devices

There are four methods, one for configuring system devices and the other for serial devices: according to hard limits, and according to soft limits. The first of these has been discussed in Chapter 5 and the second is similar.

We can summarize all these methods as shown in Figure 6–1. The goal in each is of the form configure_device(Device, ...). The preconditions check that we can safely configure it in this way; *i.e.* check that it is not already configured in some other way, or that there is no cable already attached to the device which precludes configuring it this way. Next, we have to check that there are slots available. If not, then this method with fail to be applicable and some other method (say, to configure an interface) will apply instead. The effects of the method reduce the

number of slots, by nominally attaching the device to the slot (this increases the length of the list of devices connected). There are no output goals.

6.6.4 Configuring connective components

The method `configure_ic(Interface,CardCage,Config)` is for inserting interface cards and other connective components in a card cage. Its preconditions check that such a slot is available, similar to the previous two methods. The method preconditions and effects are similar in spirit to those for configuring devices. The two methods, using hard and soft constraints respectively for deciding whether there are slots available, are described in C.6.2.

6.6.5 Configuring devices for specific needs

There are methods for generating devices to meet the requirements of particular goals: for example, disk drives to provide at least the specified storage capacity. Methods in C.6.2 are `disk_storage/3`, `total_disk_storage/4`, and `tape_capacity/3`. The preconditions for these methods check that the need has not already been met, and, by matching the input goal, that there is a goal of the required type. Note that the goal `tape_capacity(Cap,Tps,Config)`, for example, can arise in two ways: either as an original specification goal, or as an output goal from a submethod which calculates the amount of backup storage needed for the configuration, based on the disk storage capacity of the system.[1]

6.6.6 Matching attributes

The method `match_attributes(Config,Goal,Device)`, where `Goal` is of the form attribute(Config,Device,Value), is used for finding devices which possess given attributes. The preconditions of this method say that the input goal should be

---

[1]...a rather crude measure.

```
method(connect_cable(Device,Cable,C),
  cabletype(Device,Cabletype,C),
  [needs_ic(Cabletype,Ictype),        % find out connection type
   \+((guess_portgroups(C,Pgs),       % each device has exactly one
       exists_member([G,L],Pgs),      %  connection so check it
       exists_member(Device,L),       %  hasn't been assigned already
       \+type(G,Ictype)
     ))],                                          % actual cable not
  [guess_type([Device,Cable],cross(_,Cabletype))],%  of interest
  [],
  connect_cable(Device,Cable,C)).
```

Figure 6–2: The *configure_cable/3* method

of the form `<attribute>( ... )`, where `<attribute>` is one of those found in `config-attr.pl` for a single component. The effects instantiate the principle functor of the component; for example, if we have a specification

```
printers(Config,[P1,P2]),
cps(Config,P1,450),
... ... ...
```

then after `match_attributes/2` is applied, `P1` will have been instantiated to `'2235A'(N)`, where `'2235A'` is a model of printer which has a speed of more than $450cps$, and `N` is a variable used to index the printer, *i.e.* distinguish it from all other `'2235A'` printers in the configuration.

6.6.7 Cables

It might seem surprising that the planning stage should concern itself with cables. In fact, it is only the *type* of such cables which is of interest at this stage. Appendix C.6.5 explains the motivation for this. The method itself is reproduced in Figure 6–2. The goal to be proved is that a cable exists for the given device of the form `<component-number>(_)`. The preconditions ensure that this will be unique and

pick up the type of the cable. The effects do nothing material; on execution, the tactic will do all the extra work of configuring the cable itself.

6.7 The planner

6.7.1 Implementation of the planning mechanism

The implementation of the planner is very similar to CLAM's depth-first planner (van Harmelen, 1989), but I should describe the use of the input slot in matching goals in some detail since this is somewhat different. The following paragraph explains the mechanism for simple methods; it is a little different for supermethods.

In CLAM, the input is a sequent $H \vdash G$, and is rewritten to a new sequent for the output slot. The configuration conjecture, on the other hand, is a conjunct $G_1 \wedge G_2 \wedge \ldots \wedge G_n$ and, although the goals are not independent in that the proof steps for one conjunct may well affect the proof of the remaining conjuncts, nevertheless we often want to solve one before moving on to another. We would not want necessarily to solve them in the order in which they appear, however — this is arbitrary and at the discretion of the user. Thus the planning mechanism employs a smart matcher (really a unifier) which compares the input slot with the current goal, so for instance if the input slot of the method is *Input*, and the current goal is $G_1 \wedge G_2 \wedge G_3$, and if *Input* unifies with, say, G_2, then the method is applicable.

If, and only if, one of the input conjuncts unifies with the input slot of a method, then the preconditions of that method are tested, using any instantiations caused by unification of the goal with the input slot, and only if all preconditions are true is the method said to be *applicable.*

Then, as is explained in more detail in Appendix C.7, the *effects* of the method are run, and the final output of the method is obtained *via* unification during the process of testing the preconditions and running the effects. In general, less effort is used in running the effects of a method than in executing the corresponding

tactic, and it is this which makes planning worthwhile despite the apparent extra overhead.

The output goal(s) *replace* the input goal. If the method is a terminating one, *i.e.* its output slot consists of the empty list, then effectively the input goal is *removed* from the conjunct; otherwise it is replaced. The replacement goal(s) can be seen as "simpler", or lower-level: for example, the goal to configure a list of devices is replaced by goals to configure individual devices; the goal to provide a particular function is replaced by a goal to configure a particular device (or list of devices).

This is repeated as long as there are goals to be proved, until there is only one goal, `legal(Config)`, left to prove. One or more applicable methods will be found to prove this goal also: then the synthesis will be complete.

The planner implemented uses depth-first search; the first applicable method found will be added to the plan. If the planner reaches an impasse, it backtracks and the next alternative applicable method found at the most recent choice point. If a plan is found, then the planner can be asked to find another, which again it does by backtracking.

No other planners have been considered to date. Breadth first was rejected as it would cause resource problems. Iterative deepening could be used: this can be thought of as emulating breadth first search without the space overhead. The possibilities of these and of a best first search are discussed in Chapter 7, Section 7.8.2.

6.7.2 Ordering of methods

With such a deliberately simple top-level planning mechanism, it is obvious that the order in which methods occur in the methods file could be of paramount importance. In practice, I found that this was indeed the case, and thus I had to be quite careful in deciding this order. I shall now explain the ordering, which to a large extent was decided empirically, and the rationale behind it. In particular, I should explain why certain methods appear early in the methods base, thus

ensuring that they are tackled first, if applicable. This order does not always give the optimal solution, but will tend to do so in most realistic examples.

The default order of methods is as follows:

1. Configuring the processor.
2. Matching an attribute to a device.
3. Connecting a cable.
4. Configuring a device (with preference to obeying heuristic guidelines).
5. Configuring a connective component. (with preference to obeying heuristic guidelines).
6. Configuring a list of devices.
7. Meeting functional requirements of the configuration (as a whole), *e.g.* storage.
8. Checking the legality of a configuration.
9. Adding devices to ensure the legality of a configuration.

Configuring the processor

Configuring the processor constrains the search space considerably, so if this is done early synthesis will be much more efficient than would otherwise be the case.

Matching an attribute to a device

The goal

$$lpm(X) \geq 1200$$

specifying that the speed of X is greater than 1200 lines per minute will synthesize one of the set of printers capable of this speed. This is at one remove from specifying a printer explicitly but still the search space is not huge.

A general heuristic is that is is better to solve the more restrictive or restricted goals first. By the time this is done, anything which can be pinned down to an explicitly-named device will have been; this may affect the means of connection and therefore

the consumption of later "resources"[2] within the configuration, thus imposing constraints on the rest of the configuration task early on, which is desirable.

Meeting functional requirements of the configuration

On the other hand, specifying a total disk storage of 5500Mb gives rise to a large search space since such a specification can be satisfied in many ways. This is an example of a method which deals with a functional requirement of the system as a whole. There may be many ways of satisfying it on its own. However, by the time we deal with such goals some devices which could help meet this functional requirement may be already in place. Therefore this method is placed late.

Connecting a cable

The choice of cable is important from the *user's* point of view in two cases.

1. In the case of printers, connecting *via* a system cable turns the printer into a "system printer"; connecting it *via* an RS-232 or an RS-422 cable turns it into a "serial printer". The details of how this affects the utility of the printer were given in Chapter 3.

2. In the case of terminals and serial printers, the method of connection again affects the utility: in particular, how far away the devices can be located and the speed of communication.

If these factors are important, the user will wish to specify them, and taking account of these choices by the user will pin the devices down early on.

So, for example, if we have a specification

$$\begin{aligned} & printers(c) = [X] \\ & \wedge\ lpm(X) \geq 1200 \\ & \wedge\ connect\_cable(X) = hpib(\_) \\ & \wedge\ \ldots \end{aligned}$$

the process of synthesizing the (at first totally unknown) printer X is as follows:

---

[2]by which I mean slots, channels, *etc.*

1. X is added to the list of system devices (attached to a nominal cable: at the planning stage we do not actually synthesize this cable)

2. X is instantiated to (say) `'2932A'(_)`: a printer with the required speed.

3. Later the `configure_device/3` method will ensure that there is a slot for the other end of the cable in an HPIB channel.

Configuring a device

The methods for configuring a serially-connected device and for configuring a system device according to heuristic limits are relatively early in the methods database. Having a "floating" device represents a commitment to try to configure that device and it makes sense to tidy up the connections rather than trying other methods which generate yet more devices to be configured. However, it is not always the case that this way of going about the configuration task gives the "best" solution, in terms of either efficiency or cost. However, there are supermethods which control the use of this method quite effectively, of which more will be said later (Section 6.8.2).

Rather than configure a device according to heuristic limits, using the method *h_configure_device*, we may instead use *configure_device* to configure it according to legal limits. This latter method is placed later in the methods base. Use of *configure_device* is only really sanctioned if *h_configure_device* failed to apply and there was no way round the problem (*e.g.* selecting a different set of devices).

Configuring a connective component

This must be placed after the method to configure devices according to heuristic limits and the method for configuring serial devices. Only if these have failed to apply do we want to consider configuring extra interface channels *etc.* Thus, irrespective of any explicit strategy which may be encapsulated in a proof plan, these methods are naturally considered only when strictly necessary.

Configuring a list of devices

This deals with goals of the form

- `printers(Config,[<list of printers>])`
- `terminals(Config,[<list of terminals>])`
- *etc.*

where the devices, of whatever type, are either given explicitly, as in:

- `printers(Config,['2235A'(_),'2680A'(_)])`

or implicitly, by conjunction with goals specifying attributes which the devices must have, as in:

- `printers(Config,[P1,P2])`,
- `cps(Config,P1,400)`,
- `ppm(Config,P2,40)`, ...

Now the order of the methods found by the planner means that:

1. First the method `match_attributes/2` is used twice, once on the goal `cps(Config,P1,400)`, and once on the goal `ppm(Config,P2,40)`, to find an instantiation for `P1` and another for `P2`.

2. The method `configure_device_list/2` is applied on the goal `printers(Config,[P1,P2])`, and the output goals from this method represent obligations to configure the devices `P1` and `P2`.

3. The method `configure_device/3`, which is terminating, is applied twice.

Checking the legality of a configuration

The only methods after this in the methods database should be those which are invoked if a configuration is *not* legal, *i.e.* which are tried in the event of the `is_legal/1` method failing to apply.

Adding devices to ensure the legality of a configuration

The last methods are those for synthesizing various vital components, such as system disks and backup, which may well be specified explicitly by the user, but may equally well not be.

6.8 Ensuring good design

6.8.1 Managing heuristics

We can see instances of heuristic, or soft, limits in Appendix C.4 and the equivalent hard limits in Appendix C.3.6 Let us compare two methods for configuring channels:

1. according to soft constraints and
2. according to hard constraints.

They are shown in Figures 6–3 and 6–4 respectively, omitted some housekeeping preconditions which are the same for both methods.

The difference between the two methods is in the use of the precondition `h_slots_available/5` or `slots_available/5`, which use, respectively, predicates `heur_limit/3` and `limit/3`.

We can try to "prefer" to use heuristic limits either:

1. By trusting to the ordering of the methods.

2. By incorporating supermethods which encapsulate strategies for constraint relaxation, using the methods for configuring devices as building blocks in these supermethods .

Supermethods are discussed in the next section; the two alternatives are compared and this issue is discussed more fully in Chapter 7.

```
method(h_configure_device(Device,Interface,Config),
    configure(Device,Interface,Config),
    [   ...   ...   ...
     h_slots_available(Config,Interface,Type,Interfacetype,N),
     N>0))],
     [reduce_slots(Interface,Device,Type,Interfacetype,1),
        ...   ...   ...   ],
     [],
     h_configure_device(Device,Interface,Config)).
```

Figure 6–3: Configuring an interface using soft constraints

```
method(configure_device(Device,Interface,Config),
    configure(Device,Interface,Config),
    [   ...   ...   ...
     slots_available(Config,Interface,Type,Interfacetype,N),
     N>0))],
     [reduce_slots(Interface,Device,Type,Interfacetype,1),
        ...   ...   ...   ],
     [],
     configure_device(Device,Interface,Config)).
```

Figure 6–4: Configuring an interface using hard constraints

6.8.2 Encapsulating basic strategies as supermethods

In order to improve search control, which was limited using only the simple methods presented so far, I implemented a number of supermethods. The main problems to be overcome are

- Eliminating unproductive search and backtracking through methods.
- Incorporating strategies for constraint relaxation.

As I said in Chapter 5, there is a discernible pattern for a high proportion of configuration tasks, and this is encapsulated in the supermethod `basic_config/1`

(Figure C–13). In order to "recognize" that we are in such a pattern, we look at the input conjunct. Notice that, unlike the case of simple methods, we need to consider (*i.e.* pattern-match) on more than one of the input-conjunct goals. If the specification says something about terminals, disks, and printers, and if, in addition, the processor is specified, then we know we are in a straightforward example of a "normal" synthesis. The `basic_config/1` supermethod deals with such cases. It has no other preconditions — matching with the input slot is all that is required — and it works as explained in Appendix C.6.2, running methods, or submethods[3], in its effects slots in order to synthesize the different parts of a configuration meeting the specification. The supermethod `basic_config/1` is not, of course, applicable to simple partial specifications — such as those used for testing the object-level. However, the preconditions for `basic_config/1` quickly fail in these cases and a custom-built proof plan is obtained by the planner, exactly as before: `basic_config/1` augments the methods base, rather than replacing existing methods. It is placed earlier than its component supermethods, to ensure that it will be tried first.

The first solutions found by using `basic_config/1` are identical to those found without this supermethod. A discussion of how the use of supermethods affects search more generally can be found in Chapter 7. For now, note that the use of supermethods obviates, to a large extent, the need to ensure that methods appear "in the right order". Following on from the point made towards the end of Section 6.8.1, a more directed way of attempting to configure components using heuristic limits by preference is to write a supermethod whose effects involve running submethods to configure devices according to a pattern which was discovered by hand-working many examples: the use of the predicate `applicable_subm/3` within the effects slots forces the planner to consider the methods in the desired order. It should be stressed that this is only a "heuristic" strategy for trying to make sure heuristic limits are used whenever possible.

---

[3]Submethods are methods which can *only* be accessed through the effects of supermethods, and not directly by the planner.

6.8.3 Adding strategies for cost specifications

I looked at only one strategy to do with dealing with cost specifications, namely that of finding a configuration which kept under a given cost ceiling.

To do this, we used annotation of the configuration term within methods. This was only done when the goal $cost(c) \leq x$ appeared in the specification. There is an extra overhead in dealing with costs and so this is not done where unnecessary.

An extra argument representing cost, is added to the configuration tuple (in the first position), so that

$$C \equiv [X, Proc, \ldots]$$

and when any component is added (by the predicate `add_component/2`, the cost of the component is subtracted from X, and this new value compared with zero: if it is negative, the preconditions fail.

Thus unproductive partial configurations are pruned early on. Unfortunately, there is still a lot of search involved with some examples. More strategies could be developed, but the most likely means to success is via using information about failures in the planning process to "re-route" the planner in particular directions. This is discussed in Chapter 7, Section 7.8.4.

6.8.4 Search reduction

The search reduction performed by the use of tactics and methods was considerable. It should be noted that this is due to the desirable pruning of unwanted solutions as much as to the use of planning. For example take a simple specification for a configuration with at least 500 Mb of disk storage. There are three different models of disk drive with capacity 571Mb. One of these is connected via an fibre optic link to an HPFL chanel and the other two use system cables connected to an HPIB channel. There are seven models of disk drive with capacity less than 500Mb but such that two together will give the desired capacity. One of these is connected via fitble-optic links, the others all via system cables.

Thus it can seen that, even searching at the planning level alone, and using tactics which will prune all mixed-disk configurations, there are 10 different combinations of user disk drive alone. If we were to allow a mixture of disk drives, then there would be 7×6 *more* user disk configurations. If in addition we allow the system disk to be of a different model there are altogether $5 \times (3 + 7^2) = 260$ different combinations.

There are three different possible tape drives. This gives altogether 780 possibilities.

Now imagine that the search took place at the object-level. For each device, we would need to carry out the following steps in order to configure it.

1. Search the partial configuration to see whether the device is already provided with a cable.

2. If not, generate (synthesize) a new (unique) cable of the right kind.

3. Check to see whether the device is aleady connected to a channel.

4. If not, find a free slot.

5. Carry out *all* constraint checks: that the number of devices for that channel is not exceeded, that the limit for devices of this type is not exceeded, and that there are no incompatibilities with devices already connected.

6. Check that the device×cable×interface term is well-typed.

A useful comparison is that there are 461 inferences involved in configuring the first device at the object-level, as opposed to 76 at the meta-level. As more devices become connected, it becomes increasingly difficult to find slots for them and this ratio increases in general.

6.9 Summary of implementation

I have implemented a system which can deal with specifications as exemplified in Appendix E. Using supermethods, the search for solutions can progress in a more directed way, and the system can deal with global design goals such as those involving stipulations about cost of efficiency.

CLEM was implemented in Quintus Prolog to run on a SUN workstation. There are approximately 5,500 lines of code. It has all the predicates defined in Appendix C. It was tested systematically as indicated in Appendix E: all tests were drawn up by myself in line with the training example available to me and customer specifications drawn up for other HP systems, as it was not possible to get real 950 customer specifications.

Various alternative formulations and an analysis of performance will be presented in the next chapter.

Chapter 7

Implementation Issues

A nice adaptation of conditions will make almost any hypothesis agree with the phenomena. This will please the imagination but does not advance our knowledge.

J. Black

7.1 Introduction

Chapter 3 described the configuration task and Chapters 4 and 5 proposed a theory for the domain and some proof plans for configuration. In Chapter 2 I described some of the shortcomings of existing configuration software, and Chapter 6 described the CLEM system for configuring HP3000 series systems. In this Chapter, I will discuss various issues involved in implementing this system.

The main issues to be addressed are:

- Research methodology
- Representation issues:
  - How to categorize certain components.
  - How to deal with "supported" and "non-supported" (obsolete) components.
- Determining strategies:
  - Deciding on a default set of methods.

  - Mechanisms for incorporating non-default methods.
  - Learning new methods.
- The distribution of effort between planning and execution.
- Controlling search:
  - Types of planners.
  - Order of methods.
- Obtaining multiple solutions.
- Dealing with failure.
- Methodology for future development.

7.2 Research methodology

The prototype automatic configurer, CLEM, which I described in Chapter 6, is the test of the theory of configuration outlined in Chapters 3–5. Chapter 3 set out the issues to be addressed in tackling the task of configuring hardware to meet user specifications: ensuring a legal configuration (*i.e.* one which works), meeting the customer needs (conforming to the specification), and incorporating at least some of the design issues (for example, one which is reasonably priced, and runs as efficiently as possible). Chapter 4 described an object-level theory of the configuration domain; and Chapter 5 described how the proof planning technique enabled me to represent and implement the meta-level reasoning involved. Test runs of CLEM are given in Appendix E, and runtime statistics in Appendix F. The results of these tests demonstrate that CLEM does indeed produce configurations in reasonable time. It is pertinent in this Chapter to outline the research methodology used in setting up these tests, and to describe fully what is being tested, and the significance of the results.

A brief description of what is being tested is given with each test specification.

The first tests are very simple — in fact, leading to "degenerate" configurations (but correct for the specification, which is itself incomplete), and test each part of

the configuration task separately: configuring storage devices; configuring terminals; configuring printers.

Later tests enhance each of these specifications, allowing more detail to be given — for example, specifying how a terminal is to connected in terms of which type of cable is to be used; specifying that a printer is to be configured as a serial, rather than a system, device.

Next, complete specifications are tried, starting with rather small systems (to make it easy to check the output) and going on to large ones, to assess the runtimes involved and determine whether they are acceptable. These configurations may still be synthesized in such a way that all heuristics are obeyed; the tests check that this is, in fact, done as default.

Lastly, there is a series of tests which take specifications which cannot possibly be configured whilst maintaining all heuristic constraints intact, to check how CLEM relaxes these constraints in practice.

To sum up, the questions to be asked are:

1. Are the solutions the ones expected, given the implementation?

2. Are these the solutions that a configuration expert would find?

3. In any of the solutions expected and obtained, are there shortcomings, and how could CLEM be modified to overcome these (if at all)?

In addition, several different potentially acceptable implementations were tried, and tests were run in order to assess the effects of changes to various factors. In particular, Appendix F shows comparative runs for

1. Changing the balance of work between planning and execution stages.
2. Stronger or weaker preconditions.
3. Using simple methods alone, or incorporating supermethods.

In addition we need to answer the question as to whether planning is necessary at all. This question was conclusively answered, by the fact that we had to test

comparative performance with and without planning by choosing very small configuration specifications: otherwise the runtime proved unacceptable. Also, the solutions obtained were not of good quality in some cases.

7.3 Representation issues

7.3.1 Some categorization difficulties

In general, it is simple to decide on a type for each kind of component. The flat typing adopted (Chapter 4, Section 4.4.1) assists in this as it is relatively easy to decide that component X is of type **printer**: it is only at the stage of synthesizing a cross-product term (of, say, an object of type **printer** with an object of type **system-cable** that it becomes a "system device".

However, one component out of the test set proved more difficult to categorize: namely the modem, for reasons which are explained below. There is also the potential problem of new components posing categorization difficulties, which is also discussed below.

7.3.2 Modems

There are three types of **portgroup** for connecting serial devices, corresponding to the type of "connector" used. Serial devices can be connected either by RS-232 cable, or by RS-422 cable, or alternatively by modem. There is a choice of three components for attaching these connections: the RS-232 portgroup, to which 8 RS-232 cables can be attached; the RS-422 portgroup, to which 8 RS-422 cables can be attached; and the modem portgroups, with 6 modem connections. However, modems can alternatively be connected using spare RS-232 ports. This gave two alternatives:

1. • Have three types of portgroup: namely **rs232ports**, **rs422ports**, and **modemports**.

- Have three types of cable: **rs232cable**, **rs422cable**, and **modem**.
- In addition to the three obvious cross types, define **rs232ports** $\times$ **modem** to be well-typed.

2.
- Have two types of portgroup: **rs232ports** and **rs422ports**, where the modem portgroup component is a member of the type **rs232cable**.
- Have two types of cable: **rs232cable**, to include modems, and **rs422cable**.
- Define the two obvious cross types.

Either would have been acceptable for the current knowledge base. However, with an eye to maintenance, I chose the second alternative, as it seemed to reflect better the interchangeable nature of the components. In the future, one could imagine different kinds of connections would be invented, and maintenance would be easier if it were just a matter of adding new members of existing types, rather than having to add new types.

7.3.3 New components

One constant worry with any system is how technological change will affect its maintainability. In extreme cases, it may be rendered redundant. The purpose of separating object-level from procedural and meta-level knowledge was to minimize, and localize, the changes which need to be made. Even over the relatively short period when I was working on knowledge elicitation, components were being deleted from the price list and new ones added. I responded to this by taking a "snapshot" of a set of components, developing the system, and then considering how new components could be dealt with.

Clearly, a new component which is of an existing type should pose no problem. However, changes in technology mean that new types of component have to be added. Completely new types are sometimes easier to deal with than types which seem to cut across existing type boundaries.

One example is that of data storage. I defined the **disk** and **tape** type; however a newer product combines the functions of both these.

The following changes would need to be made to CLEM:

1. Add the new type, and the new component(s) to the knowledge base. Add acceptable cross-product types, so that legal connection of the device(s) are assured.

2. Allow the device type to appear in specifications, in the same way that CLEM allows $disks(c) = \ldots$, $tapes(c) = \ldots$ at present.

3. The current methods are adequate to deal with devices specified in this way (namely using $configure\_device\_list$, $configure\_device$) so no action is needed here.

4. Allow the *function* of these devices to be specified,
   e.g. $general\_storage(c) \geq \ldots$.

5. Write a new method for matching such a specification to appropriate devices, analogous to the method $total\_disk\_storage$.

6. Write a "higher-level" method which enables such a device to be synthesized in answer to less explicit needs. This is explained below.

With these additions to CLEM, requests for such devices can be made explicitly in specifications; and we can request a total storage capacity to fulfill all our storage, swap space and backup needs. However, in addition, if step 6 is taken, users can specify only primary user storage requirements, and have other needs (backup *etc.*) supplied automatically using the new device, as an alternative to the more traditional disk-tape combination.

I have not implemented such a higher level method in CLEM, as it needs expert knowledge relating to how storage needs are assessed or calculated, but such information is potentially available and could be used to update CLEM. Implementation is analogous to the present method for calculating backup needs.

The steps enumerated should be compared with what is involved with maintaining a large and complex traditional rule-based system. Although there is greater

overhead in setting up CLEM, this ought to be recuperated in the maintenance and debugging stages. Maintenance in CLEM is greatly assisted by these two factors:

1. The different levels involved can be advised by the different personnel affected: reflecting the division of labour between production, sales, engineering, and configuration experts. These different categories then correspond to the different groups of people who then maintain the system.

2. There should be far less likelihood of "knock-on" effects, whereby a change made in one part of the program affects completely different parts of the system.

7.4 Current and obsolete components

Dealing with obsolescence is not a trivial task and it has not been implemented in CLEM. The difficulty lies with components lying in a kind of limbo in which they will not be actively promoted by the sales representative but may be requested by the customer; in particular by existing customers who wish to extend a configuration or use familiar hardware. Thus the sales representative has to hand a list of hardware which is "supported" by a processor — for such hardware, some kind of guarantee will be given that it will function properly and that the company will maintain it. However, in order to ensure the goodwill of the customer old components must also be supported to some degree. An automated configuration system must also know about such components, in case they appear in specifications, but should not "volunteer" them if they are not explicitly requested.

On the other hand, there are many components which are not at all suitable for particular hardware systems, and these must be disallowed from appearing in specifications — I mean here that such specifications must be rejected (helpfully) by the system. In the current prototype of CLEM, there is no validation procedure for the latter problem, but such a routine would be trivial to implement.

However, the question of valid but less supported devices is a more difficult one, and two alternative solutions are proposed here.

The first possibility is a more elaborate conditional type-checking: checking each component in the configuration for whether it is supported by the processor. This check could be placed in the preconditions of the submethod for adding a component to the system. The precondition could be waived if the processor is unknown; in which case a mutually restrictive precondition must be placed on the method for configuring the processor: namely, that the chosen processor supports all currently configured devices. (This shows why it is desirable to decide on the processor early on). However, it is difficult to find a clean way of distinguishing between components which are readily supported, and those which are only reluctantly supported.

The second possibility will work only if the processor is decided first. Once the processor has been configured, the configuration system will then load all components supported by this processor[1] from the main knowledge base. Hence one effect of the method *configure_processor* will be to load the relevant knowledge. Thereafter, components which are not explicitly specified and which are not on the official list of supported devices will not be inadvertently synthesized. On the other hand, if a component appears explicitly in the specification, permission will be sought by the system to access the main knowledge base. As appropriate, an error or a warning message will be given as a side effect of the submethod *add_component*.

In passing, it is worth pointing out that it would be useful if this submethod did not always fail outright because of the unavailability of a component to a system, but allowed an alternative component to be sought. Not only would this be more constructive than failing outright, but would also be useful in tackling the somewhat harder task of configuration upgrading (Chapter 8, Section 8.2.3), where, if the processor is replaced, many existing components will necessarily have to replaced. This would enhance the generality of the proof planning system.

---

[1]Note that sales representatives have lists of supported devices available to them.

7.5 Developing strategies

7.5.1 Determining the processor

Existing software for configuring computer hardware relies on the processor being known at the outset. CLEM does not need the processor to be explicitly specified, but, if it is not, the first stage is to decide what an appropriate processor would be, to configure that processor, and to synthesize a "template" configuration incorporating this processor.

The heuristic used has been the number of users. It has not been possible to explore other possible heuristics. However, the current knowledge which CLEM possesses so far seems on a par with existing commercial systems.

The supermethods (see Section C–13) which I used control search to the extent that the processor is always synthesized first, whether specified or not. In the absence of such control, and of an explicitly specified processor, there would be potential problems if CLEM were allowed to roam unfettered over the whole of the potential knowledge base. For example, disk drives, tape drives, printers, and terminals could be configured, such that there was no processor which could possibly support all of them: the backtracking through the search space involved could be horrendous.

It should not be difficult, given access to the appropriate personnel, to encapsulate the "rules of thumb" (apart from number of users) which sales representatives use to decide the most appropriate processor. One potential lead is in looking at potential "bottlenecks" — some systems tend to run out of CPU time; others tend to run out of ports; and so on — this advises us to look at the most important resource, or resources, first, and make an informed choice.

Adding this knowledge involves enhancing the method of CLEM which synthesizes an appropriate processor in the absence of an explicit request. This method presently assesses processor needs on the basis of the number of users (from the number of terminals to be configured).

This issue would be important if CLEM were enhanced to deal with *system upgrading*: sometimes the new processor will be specified explicitly; but in general the need to upgrade arises because some vital resource runs out in the existing system, in which case it is necessary to find a processor which can carry the additional load, be it extra users, extra memory, or whatever.

7.5.2 Synthesizing storage devices

Consideration of an alternative set of systems, namely the HP 800 range, tested some of the "first attempt" methods of CLEM to destruction. To understand why this is so, first note that these systems, unlike the HP 900 series, have built in storage; however, this can be supplemented by extra (external) disk drives. Now, my original *disk_storage* method had, as its main precondition, a requirement that the value of $disks(c)$ would be (wholly) variable: tantamount to "don't configure something if you have it already". The value of the input slot is the specification $disk\_storage(c) \geq x$; thus if x is greater than the value of the built in storage, this method would not be run, yet the specification would not be met.

The precondition was altered to check merely that the system did not already meet the disk storage requirements. The value of $disks(c)$ takes the form of a list with a variable tail — this remains until the close of the synthesis process — thus this tail can be instantiated at any stage.

This weakening of the precondition is useful also if the specifications are posed in such a way as to gradually give more information. For example, if at some stage it were possible to incorporate knowledge about the disk storage requirements of certain applications, then we could envisage the synthesis process going through stages thus:

1. The user asks for disk storage of at least 1.6Gb.
2. Disk drives giving this amount of storage are synthesized.
3. The user asks for 100 terminals, to be used for a particular application.
4. The system calculates that 2Gb are needed to run this application.

5. An extra disk drive is configured to allow for this.

Again, this is unimplemented, due to the current lack of availability of the necessary domain knowledge, but could easily be incorporated if sufficient domain knowledge was available.

Note that in the case of systems with built-in storage, the term for disk storage is partially instantiated as soon as the skeleton system is synthesized, following the synthesis of the processor.

Another use for this flexibility afforded by the use of partial instantiation of lists is that the casual user of the system may want to specify the model of disk drive but not know the total disk capacity required. Thus a specification $disks(c) = [7937H(\_)|\_]$ may be given, followed by the specification $disk\_storage(c) \geq 1000$ (or this goal may result from proving higher level goals, such as the need to support certain applications, as explained above); this results in a second disk drive, of the same model ($7937H$) being added to the tail of the list.

7.5.3 Limits

There are places where lists could have been "closed off". For example, the template for the 950 system consists of a list of two MUXes. It would have been possible to close off this list, since no more are allowed in this kind of system (unlike certain other systems, where MUXes are used for terminal connections, and the number needed depends on the number of ports). However, I felt that having the limit of two MUXes encoded explicitly elsewhere made for ease of maintenance, in case this restriction were lifted subsequently. Although the list of MUXes is not closed off, a third MUX will not be added in the course of synthesis, as this would violate the constraints which are checked before every such addition. Looking to further developments of CLAM, which could introduce explanation facilities (see Chapter 8, Section 8.3.2), we could imagine a dialogue as follows, directly linked to the failure of a specific precondition of a method:

"Can I add another MUX?"

— "No, because this puts you over the limit for MUXES in this configuration"

If, instead, this limit were "hard-wired" into the template-fixing, it is difficult to envisage what the basis for an answer to the question "Can I add another MUX?" would be.

7.5.4 Card cage constraints

In the case of the 950 processor there are a fixed number of slots available, although there are constraints on how they may be used; for example, even if there is an empty slot in a card cage, we may not configure an HPFL interface card there if this would mean the total number of HPFL cards exceeded three. Moreover, the values of these constraints are independent of the number of card cages in the configuration.

However, in the case of the 925, which was incorporated in order to test the maintainability of the system, this is not the case. The addition of a second card cage (only two are allowed in total) makes one card fewer available to the first card cage. In effect, the second card cage "uses" one of the slots available to the first card cage.

This affects the predicate *slots_available*/3, one of whose arguments is the (partial) configuration, with the processor instantiated. Otherwise, the system is unaffected, and the method *configure_cc* at the top level is unaffected.

In practice, more search is involved in the case where a 925 system is being configured and a second card cage required, since usually the slot in the first card cage will have been filled before it is apparent that a second card cage is needed; the step which "filled" the first card cage with an extra channel will have to be undone before the method to configure the second card cage can succeed. Fortunately, 925 systems are generally smaller than 950 systems, and both planning and execution time tend to be shorter for them.

The only other changes which had to be made as a result of adding 925 systems is additional constraints: there is a restriction of two tapes per channel; and in addition only one 7980*A* tape is allowed per channel. However, these restrictions are in the same spirit as other constraints applying to 950 systems and did not involve any fundamental rethinking. Fortunately, all predicates relating to constraints had been formulated with the (partial) configuration, and hence the processor, as one argument.

For example, the restriction on 7980*A* tapes mentioned above is expressed as:

```
not_share_with(['7980A'(_),_],[Tape,_],['925'(_)|_]):-
                    guess_type(Tape,tape).
```

where we needed to partially instantiate the configuration term to show the processor. Alternatively, we could have

```
not_share_with(['7980A'(_),_],[Tape,_],C):-
                    processors(C,['925'(_)|_]),
                    guess_type(Tape,tape).
```

7.6 Methods

7.6.1 Default method set

Simple methods sufficed for specifications in which the processor was explicit, and where no relaxation of heuristic constraints proved necessary, and in which cost did not feature. However, supermethods proved essential for all other cases in order to control search. Solutions could not be obtained in real time otherwise.

However, unless the user is completely indifferent as to how constraints are relaxed, some guidance is needed in loading supermethods. The supermethod *basic_config/1* is loaded regardless, and should be tried first unless there is a cost

constraint (for which there is a separate supermethod). However, there are alternative ways of relaxing constraints; which is tried first depends on the order in which they are loaded into the system on initialization.

It was decided (somewhat arbitrarily) that no track would be kept of the cost of a system unless a specific goal, relating to cost, appeared in the specification. The justification for this is that, in the absence of such a goal, the system should concentrate on synthesizing the most efficient configuration possible. To trigger the use of the *cost_configure* method, a goal $cost(c) \leq \ldots$ must appear in the input specification.

7.6.2 Incorporating non-default methods

Ideally, the task of synthesizing a less efficient configuration, *i.e.* one in which not all heuristic constraints can be obeyed, should be an interactive one. I am convinced that decisions about which of several constraints to relax cannot be fully automated in the near future. I therefore believe that future versions of CLEM should load in extra methods dynamically after consulting the user. The user, at the point when it became clear that no method was applicable for configuring a device (because it would involve breaking a heuristic limit), should be asked for her preferred (or least unpreferred) options. Chapter 3, Section 3.8.6 dealt with the issues involved in this.

The user wishing to experiment can load different methods; this is similar to the facility in the CLAM theorem proving system (van Harmelen, 1989).

7.6.3 Maintaining and modifying methods

The present implementation of CLEM is one in which the detail of exactly which channels are to be used for connecting particular devices are left to the execution stage. For 950 hardware systems this seems perfectly adequate in practice. In addition, the more stringent checks can be omitted. However, tests 41–42 in

- 2 tape drives, 1 system disk, and 3 printers have to be configured on HPIB channels.
- The plan
  1. configures 2 channels as part of the template
  2. configures the system disk, nominally on channel 1
  3. configures 3 printers, nominally on channel 1
  4. configures the 2 tape drives; one on channel 1 (which is then full) and one on channel 2
- However, trying to execute this plan is not successful, due to various constraints:
  1. system disk goes on channel 1
  2. the 3 printers have to go on channel 2
  3. The 2 tape drives cannot go on channels 1 because they are incompatible with the disk drive; there is only room for one of them on channel 2.

The planner eventually comes up with a plan which can be executed on its sixth attempt: an extra channel is configured (the preconditions allow this as long as there are not empty channels already around).

Figure 7–1: Test 41: replanning on execution failure

Appendix E demonstrate how a plan can be made which cannot be executed. Appendix E.2 gives a test run of this example. Test 41 is also outlined in Figure 7–1. Notice in this example that the synthesis nevertheless goes ahead, an extra channel being configured at the execution stage — thus as a "fallback position" CLEM will not fail on execution but will replan. If this is not acceptable, then the alternative version of *config_device* can be used.

The need for other modifications may well become apparent with experience of more examples. For example, the present version of *basic_config*/1 was designed to optimize the synthesis of specifications where, as explained above, without the use of supermethods, one more channel than strictly necessary was configured.

In general, the more examples are run, the better understood the heuristics for the task will be; but they are likely to remain just that — heuristics — and no

one strategy will guarantee the optimum solution in every case. The difference in cost involved, between an optimal solution and a near-optimal one, is small in percentage terms of the total cost of the configuration.

These issues are discussed more in Section 7.8.1.

7.7 Object-level synthesis

Synthesis of large hardware configurations using only the object-level theory is well nigh impossible, not simply because of the unacceptably high runtimes, but also because searching for alternative solutions yields so many trivially different configurations. This is not surprising, as we rely on Prolog's backtracking strategy, and there are so many different permutations of how the same devices can be connected that these trivial permutations are all we get at first. Moreover, it is always possible to add a new channel (say) to a configuration before configuring a device — this is a perfectly legal move — so as an alternative to configuring a device on an existing channel we obtain the solution where a new channel is synthesized — even if the existing channels are all empty. This gives some strange looking, albeit legal, configurations.

7.8 Planning and search

7.8.1 The planning-execution split

The first version of the planner explicitly allocated system devices (*i.e.* those to be connected via HPIB cables) to *named* HPIB channels. In order to do this, the method *configure-device* checked, in the preconditions, that all the constraints are satisfied for a particular channel. The method is shown in Figure 7–2.

Precondition 1 checks that the device has not already been configured as any device must have a unique connection in the configuration. The first preconditions

- Name:
  $configure\_device(Device, IC, C)$
- Input:
  $configure(Device, IC, C)$
- Preconditions:
  1. $\neg configured(Device, C)$
  2. ...
  3. $\wedge slots\_available(C, IC, Type, ICtype, N)$
  4. $\wedge N > 0$
  5. $\wedge\neg$**not_share(Device, IC, C)**
- Effects:
  $reduce\_slots(IC, Device, Type, ICtype, 1)$
  $\wedge guess\_system\_devices(C, SysDev)$
  $\wedge add\_member([Device, \_], SysDev)$
- Output:
  []
- Tactic:
  $configure\_device(Device, \mathbf{IC}, C)$

Figure 7–2: $configure\_device/3$: Mark I

check that this is the correct configuration method (there is another method for connecting serial devices). Preconditions 3–4 ensure that there are enough slots available *assuming that* the constraints on which devices may share channels with which other devices do not necessitate the addition of extra channels. In other words, this condition checks that there are at least as many slots of the required type as there are devices requiring them — but this is not, in itself, enough to ensure that an allocation of devices to slots can be made.

These preconditions are necessary in both versions. It is precondition 5 which enables the explicit allocation of a device to a named channel. This checks *Device* against every other device so far connected to IC.

The tactic is called in mode $configure\_device(+Device, +IC, -S)$. Observe the same method in a version where the value of IC is lost and the tactic is called in

mode $configure\_device(+Device, -IC, -S)$ (Figure 7–3). Only the preconditions and tactic slots are given (everything else is unaltered).

- ...
- Preconditions:
  1. $\neg configured(Device, C)$
  2. ...
  3. $\wedge slots\_available(C, IC, Type, ICtype, N)$
  4. $\wedge N > 0$

  ...
- Tactic:
  $configure\_device(Device, \_, C)$

Figure 7–3: $configure\_device/3$: Mark II

At first sight, this method also seems to be picking out, and adding the device to, a named connection IC. However, note that here this choice is only a nominal one: the precise slot which will be used is not really known (since we have not checked against other devices). Note that we do not "remember" this nominal slot in the tactic slot: the argument is uninstantiated.

The hope is that we can always find an allocation of devices to channels, using only the channels we have configured. Almost always this turns out to be the case. For example, suppose we have six disk drives, one of which (we call it SDD) is the system disk drive, and one tape drive. We have two HP-IB channels and the nominal allocations at the planning stage were these:

- SDD to channel 1
- DD1 to channel 1
- DD2 to channel 1
- DD3 to channel 2 (no more disk drives may be placed on channel 1)

- DD4 to channel 2
- DD5 to channel 2
- TD to channel 1 (channel 1 still has room for tape drives)

On executing this plan (remember the channel names are lost), this allocation happens to be the one which will be tried first. However, the last step is not permissible, since TD may not share with SDD. But we can allocate TD to channel 2 instead: so the plan succeeds.

However, suppose we had five disk drives including SDD, a tape drive TD and a printer P which may not share with any disk drive. The nominal allocations could be

- SDD to channel 1
- DD1 to channel 1
- DD2 to channel 1
- DD3 to channel 2 (channel 1 full to disk drives)
- DD4 to channel 2
- TD to channel 1 (channel 1 still has room for tape drives)
- P to channel 2 (channel 1 full to all devices)

Now we have a problem. We need to separate SD and TD:

- Channel 1: SD
- Channel 2: TD

We need to separate P from SD:

- Channel 1: SD
- Channel 2: TD,P

and we need to separate DDn from P:

- Channel 1: SD,DD1,DD2,DD3 (full)
- Channel 2: TD,P

- DD4 unallocated: need another channel but there is no tactic in the plan to configure one.

So, after a fair amount of backtracking, the plan will fail.

To ensure that the plan does not fail we add an extra precondition to *is_legal*, the final method in the plan.

The method *is_legal*, thus modified, is shown in Figure 7–4. The only difference

---

- Name:
  $is\_legal(C)$
- Input:
  $legal(C)$
- Preconditions:
  1. ...
  2. $\wedge guess\_system\_disks(C, Sys)$
  3. $\wedge groundp(Sys)$
  4. $\wedge guess\_tapes(C, B)$
  5. $\wedge groundp(B)$
  6. $\wedge$ ...
  7. $\wedge check\_not\_share\_constraints(C)$
- Effects:
  []
- Output:
  []
- Tactic:
  $is\_legal(C)$

Figure 7–4: Modified version of $is\_legal/1$

---

between this formulation of is_legal(S) and the one in the other version of the planner is precondition 7.

The way the predicate $check\_not\_share\_constraints(c)$ works is as follows: it finds all system devices to be configured on channels, and all available channels. It carries out very simple checks to decide whether there could possibly be a problem.

If there are as many channels as devices, there will be no problem and no further checks are necessary. This is the most obvious case. If there are fewer channels (say there are n channels) than devices, but there are no more than n devices involved in constraints, then again there will be no problem.

Only if the simple checks do not decide for certain that an allocation can be made is a full allocation tried for. There are in-between checks we could do, using results from graph theory, and these are reported in Lowe (1991a). However, the overhead involved was not felt to be worthwhile in the case of large systems, since it is extremely unlikely that an allocation will not succeed at the execution stage if it has passed the planning stage. As we saw earlier (Figure 7–1), even when a tactic cannot configure a device on existing channels, a fall-back position, of adding another channel, is open to it, except for very constrained situations. In the case of the smaller systems, it was not known whether there were similar constraints operable; it is a matter for empirical field testing. An alternative approach would be to dynamically alter the preconditions (see Chapter 8, Section 8.3.2).

We would expect the execution stage of the modified planner to be greater, since the work of allocating a device to a channel so that all constraints are satisfied has fallen on the tactic during its execution. In the previous version, the constraints must be checked for the given channel, but they will necessarily succeed so there is no backtracking to yield different choices for the channels. To recoup this, we would want a saving at the planning stage — not necessarily enough to recoup the overhead if only one plan is found for every execution, but at least to recoup the overhead if the normal mode of operation is to search through a number of plans.

7.8.2 Controlling search

In implementing the planning mechanism. I decided to begin with a simple depth-first planner and explore other possibilities in the light of my experience of this. In the event, there were plenty of options to investigate within the same overall search strategy, and what problems there were did not seem solvable by changing

the search strategy to, say, breadth first or iterative deepening, but were solvable by other means.

There did not seem to be much to be gained by using an iterative deepening planner. Such a planner is chiefly useful if we expect the search tree to contain infinite (or at least, very deep, unproductive) branches. Such is not the nature of the configuration domain provided the preconditions of the methods are adequate. A breadth-first planner would be even more inefficient, because so many of the branches at each stage are AND rather than OR: a "choice" as to whether to configure the system disk drive or the tape drive next is hardly a choice at all, since we must eventually do both. Thus, since it seemed adequate for the purpose, the depth first was the only planner implemented.

The use of a depth-first planner means that the order of methods in the methods base is of some importance; although in general my approach is declarative, like most Prolog programs the order of the clauses can be seen as encoding an implicit control strategy. The planner as implemented does not find all applicable methods and choose one: rather it finds the first applicable method and adds it to the sequence. Only if it fails to find any applicable method at some point, or if it is asked to find alternative solutions, may it back up and choose the next applicable method at that point.

I have tried to keep this kind of implicit control to a minimum and ideally would like to implement a best-first planner which explicitly determines which of several applicable methods is the best. I did not feel that experimenting with different planners was of sufficiently high priority compared with more vital questions so this investigation must be classified as "further work" at this stage. It is a non-trivial problem to decide how to judge what "best" is in this domain, since there are several dimensions to measure along.

Some work has been carried out to investigate the use of best-first planning for CLAM (Manning, 1992). Rather than precondiitons succeeding or failing in an all-or-nothing way, heuristic scores were attached to them. The best-first planner calculated the scores for all applicable methods, for all possible instantiations: in general, preconditions can succeed in various ways. This score was used to use

the next method to apply. It was found that in some cases shorter proofs (which were better) resulted than those obtained by the depth-first planner, and in one case the proof of a theorem was found by the best-first planner which could not be found at all by the depth-first planner.

This technique could conceivably be most useful when trying to decide between strategies for constraint relaxation. In general, we cannot choose between them — only the user of the system can do this, for the specific case in hand, as we saw in Chapter 3, Section 3.8.6. If the user wishes to express a hard and fast preference then this can be implemented by deleting (or not loading) one of the supermethods. But the user could be asked to give relative scores to the breaches, which could then be used in best-first planning. However, it is not obvious how these scores could be used in planning except in a fairly *ad hoc* way. The main problem is being able to look ahead. For example, unless the user gives a very high weighting to one as opposed to another, four breaches of one heuristic will be worse than one breach of another. However, breaching the card cage heuristic limit gives some "breathing space" when not only the current device, but two or three more, may be configured without again breaching any heuristic limits. This is not true of the channel heuristic. However, if the latter is generally preferred, we need to know how likely further breaches are. This could be done by examining the structure and size of the input specification. In order to pursue this avenue, which seems the most promising, some kind of lexiographic ordering may be better than a numerical score. This merits further investigation, especially as the idea of a best-first planner for informed search is intrinsically attractive.

However, to summarize: the most effective means of controlling search found was *to explicitly encode strategies* as proof plans, encapsulated in a few supermethods. If these are sought first (as they are by the planner) then the order of methods which they use as submethods is immaterial.

7.8.3 Obtaining multiple solutions

Very rarely do users or customers want to generate just *one* solution. Usually, for any specification, they are interested in looking at alternative configurations, and choosing between them. To be of any value, the alternative solutions presented must be significantly different, and not merely trivial permutations of each other. At the conclusion of Chapter 5, I said that proof planning allowed a *useful subset of solutions* to be generated. So I am claiming that one benefit of the proof planning approach is that the sequence of solutions obtained should be "better" than those obtained by blind object-level search.

It was certainly true that the search at the object-level was not very efficient, as shown already. It was also true that the process of searching for further solutions, once one had already been found, was not at all satisfactory. The question was whether searching the planning space would be better, and this indeed has turned out to be the case. We shall now appraise how far the use of these methods solves the problem of searching for alternative "good" solutions.

Consider the sample run, using only the object-level theory, of the simple example shown in Appendix E.3, showing more than one solution being obtained. There are two drawbacks.

1. The first three solutions take the same three devices in different permutations: the two disks on channel 1 and the tape on channel 2; the tape and user disk on channel 1 and the system disk on channel 2; the tape and user disk on channel 2 and the system disk on channel 1; and so on. We do not want 'alternative' solutions of this nature.

2. There is no difference in status between hard and soft constraints, so we cannot manage controlled constraint relaxation at all.

Now contrast this behaviour with that of the planner. This behaviour is exhibited at the start of Appendix E.3, and demonstrate search at the meta-level. The first three solutions *are* non-trivially different, representing a real choice of options for the user.

7.8.4 Dealing with failure

Destructive effects of failure

Previously we discussed cases in which a plan fails on execution, or does not faithfully carry out the plan due to the absence of a vital component at the crucial time. Another possible kind of failure is where the planner fails to find a plan when one should be possible. The balance of probability between these two kinds of failure depends on how strong the preconditions are. The current implementation errs on the side of weak preconditions: stronger preconditions, of the form given in Section 7.8.1 would guarantee a specific slot for every component, but we have already seen that this is inefficient. The worst that can happen in all *practical* situations for *large* systems is that we are unable to configure a small number of devices (probably only one) according to soft limits. For in all practical situations, the configuration will be nowhere near its *legal* limits: it will be within its heuristic constraints or a little outside. There is plenty of scope for patching the plan to find a legal solution which, as we have seen (Figure 7–1) can often be done automatically at the execution stage. Whether the configuration thus found is acceptable to the user is another matter. But this, in turn, can be amended — in the example of Figure 7–1 by simply deleting the extraneous channel, for example. In passing, it should be mentioned that one possible solution to the problem of channels being configured too late in the plan is to reorder the plan so that all channels are configured first. This could be done more elegantly by using a supermethod in which the first submethod is an iterative method to configure N channels. N is instantiated by the end of the planning stage. Note that such an artifact can only be used as a submethod; not only must we be able to instantiate N (by using the the checking mechanisms in the preconditions of the *configure_device* method) but the unfettered use of the method which configured an uninstantiated number of channels could be disastrous.

Constructive use of failure

So far, no constructive use has been made of failure at either the planning nor execution stage. Ireland (1992) describes how critics could be used in CLAM and such techniques could greatly aid the efficiency of configuration synthesis in certain notable cases.

One example of this is where a configuration must be synthesized which keeps to a budget. In this case, the most likely cause of failure at a branch is not that a component cannot be physically accommodated, but that the addition of such a component violates the cost constraint.

Here, use could be made of the fact that it was the cost precondition that failed. A local "patch" could be tried: look for a cheaper component which meets the need. If this patch fails (in practice, this is probably going to do no more than postpone the problem), then places need to be found earlier in the plan where cheaper components can be substituted.

For example, suppose that fibre optic disks were chosen. These are expensive and could be substituted by HPIB-connected disk drives. The critic could suggest this.

If these disk drives were not randomly chosen by the system but insisted on in the specification, the planner must fail, but could inform the user that the failure was due to the violation of cost constraints and suggest alternatives to the specification: cheaper disk drives (or a higher budget).

7.9 Methodology for future development

CLEM is only a prototype system; in order to be used "in the field" it would need several enhancements, notably to its interface. It also needs to be brought up to date.

In pure research terms, these are trivial issues; however, they are necessary if others are to be able to use the system effectively.

One important issue has emerged. If proof planning is to be effective in producing desirable solutions, whether this means well-designed hardware systems (in the domain of configuration) or elegant proofs (in the theorem proving domain), a true understanding of the domain is a vital ingredient in developing such a system. This does not mean one has to be a configuration expert; but there are no short cuts to be taken. It is not enough to learn a few superficial rules without also understanding what is good or bad about various alternative solutions, and striving to understand or develop strategies which return good solutions more often than bad.

Thus in order to enhance CLEM's configuration strategy it will be necessary to gain more knowledge. Various indications of this have already been given: for example, knowledge of how to relate disk storage requirements to applications; understanding of the reasons behind various heuristics.

As far as the more exciting, and domain independent, research issues are concerned, the one which seems most likely to increase the power of the system is harnessing failure to constructive ends. This is perhaps even more important for this domain than for theorem proving. In particular, failure in configuration can be more fundamental than simply searching the wrong branch of a proof tree: it can follow necessarily from impossible specifications. In this case, the system needs to be capable of a useful dialogue with the user to resolve differences and modify the specification.

The key to this is the use of preconditions to provide explanations for the user, and suggest remedial action. I hope to explore this issue in the near future.

Chapter 8

Further Work & Conclusions

There could be no fairer destiny for any ... theory than it should point the way to a more comprehensive theory in which it lives on, as a limiting case.

Albert Einstein

8.1 Introduction

My aim in this thesis has been to explore an alternative paradigm for knowledge based systems. To this end, I have applied the proof planning approach to configuration problems.

With this as my aim, I studied the domain of configuration with the help of researchers working at Hewlett Packard Research Laboratories, and other people involved with configuration on a day to day basis. I devised a theory of how computer hardware configurations can be synthesized from specifications. In order to test my ideas, I designed, implemented, and systematically tested an automatic configurer, CLEM. Chapter 7 discusses the tests, the details of which are given in Appendices E and F.

In Chapter 1, I gave various criteria on which the success or otherwise of such a venture could be assessed. Established systems, such as XSEL, already exist for the configuration task, but I believed them to be deficient in various ways, as explained

in Chapter 2. In particular, I wanted CLEM to be amenable to adaptation and change, and for its object-level knowledge, heuristics, and strategies to be explicit and transparent. Another aim was to demonstrate a methodology, within the proof planning paradigm, which could be transported into other knowledge based systems tasks.

I shall now assess how far I have achieved these goals, and what could be done in the future to consolidate this work. However, in describing the progress of a prototype system, it also remains to ask whether this alternative approach could ever be feasible in practice, given that it is fundamentally different from most traditional expert systems.

This summary of what I have achieved with CLEM will be conducted with particular regard to:

- Its performance and potential in comparison with other systems, with especial consideration given to its flexibility, particularly in the face of technological change.
- Its maintainability.
- Its potential for adaptation to new problems in the same domain.
- Verisimilitude: its performance in comparison with an expert (human) configurer.
- Perspicuity: how easy it is to understand what it does, and how, and why.

In recognition of CLEM's limitations, I will also set out the most promising possibilities for extensions to the work already done.

In addition, I will assess how transportable this approach and the proof planning techniques are, in general, to other IKBS domains.

Lastly, there is a discussion of the prospects for adopting the proof planning approach in general.

8.2 Assessment of CLEM

8.2.1 Comparison with other approaches

CLEM should not be assessed only in isolation, but also in comparison with other configuration systems. It is interesting to step up a level, taking, for example, XCON/XSEL as a prime, and well-established, example of the *genre*, and CLEM as an example of what could be achieved using my approach, and compare the more traditional expert systems approach with the proof planning one. There are similarities in the course of development of both systems. For example, I too needed to start with a subset of components and a subset of the tasks to be performed. I needed to learn from experts and manuals much about configuration procedure and strategy. Some of my early solutions proved to be faulty in that I had overlooked certain factors.

However, XCON developed by accumulating its knowledge in a rapidly growing and unwieldy rule-base. Like the proponents of frame-based and semantic net systems, I too recognize the need to store knowledge "in the right place", but in my case this manifests itself, not in hierarchies of objects with procedural dæmons attached, but in a clean separation of different *kinds* of knowledge: object-level theory, heuristics, and control knowledge. It is this separation which gives the proof planning approach the potential for growth, not merely in terms of adding more components to its repertoire and assimilating constant changes to its rules, but also in the sense that it should withstand more fundamental assaults: changing technology being the most problematical for traditional systems.

8.2.2 Maintainability

Examples of the kinds of problems faced in maintaining an automatic configurer were given in Chapter 7, Section 7.3.3. Any student of systems analysis knows that maintainability is an important criterion for judging the success of a computer

system. Usually this question is assessed with regard to systems which have been in use, in the field, for some time. This clearly has not been done with CLEM, which is merely a prototype. However, CLEM was not born, complete and perfectly formed, but evolved over a period of many months. Its expandability was, therefore, an important issue right from the start, and not simply something to worry about for the future.

It has proved possible to extend the control knowledge of the planner on an incremental basis. In Chapter 1, I described the benefits of such an approach.

With CLAM, unlike many knowledge based systems, the development overhead is considerable as it requires a good understanding of the domain and the problem. However, I claim that this is a benefit in the long run. The developer of the system is forced to make knowledge explicit and modular. This may well result in an overall saving in programmer-hours.

My inexperience, of configuration and of developing knowledge based systems generally, showed up in the early stages. For example, maintaining the knowledge base by bringing in different processors seemed to defy the existing framework. But a deeper understanding of the task, the result of being obliged to formulate the domain in terms of logic, highlighted the essential generalities of the configuration task. The recasting of the problem and the logical framework which was developed as a result proved sufficient for all later developments and additions.

8.2.3 Tackling new problems

In Chapter 1, Section 1.3.2, I claimed that the same object-level theory should be usable for different problems in the same domain; such is the flexibility of the proof planning approach. I will suggest two such tasks, and indicate what is involved in implementing proof plans to carry them out.

Order checking

Once the sales representative has decided on the configuration, he draws up an order — that is, a list of order numbers. This is not as straightforward a task as it might appear. An order number might refer to

1. A single component — *e.g.* a tape drive; a cable.

2. Two (or more) related components — *e.g.* a disk drive and the cable needed to connect it.

3. A *bundle* — *e.g.* a cabinet which holds eight disk drives, and the eight disk drives themselves.

4. A minimum configuration — *i.e.* in the case of the 950 processor one order number covers the processor, a certain number of memory boards, two card cage adapters, two MUXes, one LANIC, two HP-IBs, *etc.*

It is difficult to produce orders which are:

- Correct. Confusion may arise if, say, the disk drives are always bundled with their cables but the tape drives never. Mistakes made are either forgetting to order a cable, or ordering an extra one.

- Optimal. If the configuration requires ten disk drives, is it better to order a bundle of eight and a bundle of two, or five bundles of two, two bundles of four and a bundle of two, or even three bundles of four (which over-configures)? This is not straightforward due to the non-linear cost functions employed in competitive markets.

It seems desirable, either to check orders drawn up by sales representatives, or to automate this process completely.

The object-level theory necessary for this task is exactly that used by CLEM. Moreover, the same heuristic constraints apply. The major difficulty faced in this task is how to synthesize a configuration, using *more or less* the components provided, in the absence of a specification. One possible strategy is the following:

1. Extract the main components, *i.e.* the processor and devices, from the list of components given. This involves preprocessing, before the planning stage. For example, components involving tape drives are gathered together in a list, T, say, and the specification goal resulting has the form $tapes(C) = T$. The components not assembled into this specification are held in a components list.

2. The main methods for synthesizing a configuration from this specification are unaltered, except that the method language predicate *add_component/2* is rewritten so that it attempts to find components from the components list (and deletes them from the list, once used). It can, if necessary, synthesize a component not on the list, printing an appropriate warning message to the user.

3. Finally the methods for synthesizing important parts of the configuration such as backing store are run and the results compared with what is actually configured — the user can be warned about any shortfall. For example, if the capacity of the tape drives configured is C_1, but the capacity required calculated from various attributes of the configuration is C_2, and $C_1 << C_2$, a warning message can appear.

The main alterations to CLEM are the addition of a preprocessor, a new super-method which incorporates *basic_config/1* with the advice method needed for stage (3) above, and a rewriting of the *add_component/2* predicate.

System upgrading

System upgrading seems amenable to transformation techniques. If the customer wishes to expand beyond the capabilities of her current hardware configuration she may consider a system upgrade. There are three kinds of upgrading:

1. Retaining the current processor.

2. Board upgrades: enhancing the current processor.

3. Box upgrades: replacing the current processor.

The first decision to take is which of these is possible. Then, we need to see what new components are needed.

In effect, we have an old configuration (which met an original specification: this may have been lost) and a new specification. We want to add new components (and we may need to delete others) so that the new specification is satisfied.

Obviously, one way of going about this task would be to replace everything: this is undesirable. Rather, we would wish to retain most of the old configuration, especially those parts which are expensive (unless perhaps they were obsolete).

Another way would be to find out whether it was possible to retain the old processor. Next, whether a board upgrade might be possible. Finally, we need a kind of *middle-out reasoning*, in which we start with some of the components, and try to determine which processor would be best. Then a new configuration based on the old one is synthesized.

Incorporated into the general strategy would be a supermethod similar to that for order checking, but with the devices also relegated to the components list, for possible rejection or replacement by the predicate *add_component/2*.

8.2.4 Verisimilitude

Provided a plan can be executed successfully, the resulting configuration is guaranteed to be legal, *provided* the object-level theory is correct and complete. Naturally, if there are constraints missing from the object-level theory, we cannot guarantee soundness.

However, the object-level theory is coded declaratively, which makes it easier to check:

- that components are correctly assigned to types;
- that there is a complete list of components;
- that all constraints (hard and soft) are included.

Since the tactics make use of the object-level theory, which is sound, they too will necessarily result in legal configurations. Using tactics means that *completeness* will be lost. However, no expert would provide *all* solutions to a specification, even for very trivial cases — we saw (for example, the example in Appendix E.3 referred to in Chapter 7) that many permutations exist, where the same components are configured in different orders, *etc.* Thus completeness is not necessarily a goal for knowledge based systems. In fact, in practice it is better to forgo completeness than allow the user to be overwhelmed by trivially different solutions.

Moreover, many solutions which are technically legal, even if they were not trivial permutations of other solutions, would be frowned upon by experts. For example, using a different model of disk drive for storing the operating system from those used for user disks would be thought odd. Where we need to use tactics to fill in gaps in the explicit specification, these are based on the same design considerations as seem to be used by experts.

The position I have taken can be summarized thus:

1. The user (customer) is always right — any specification which is legal must lead to a synthesizable configuration by the automatic configurer.

2. Any free choice left to the automatic configurer should lead, if possible, to the synthesis of well-designed configurations

The solutions found in testing conformed to this agenda. Note, however, that for large systems, even with this kind of incompleteness there will still usually be a very large number of different solutions. The user not wishing to plough through all of them would be well advised to scan through a few solutions (at the planning level only) and retry, making her specification more detailed on the second pass. This kind of negotiation is common in practice, between sales representative and customer, and cannot presently be emulated by CLEM, which is completely automatic and in no way co-operative. However, proof planning as a paradigm gives better prospects for co-operative systems than existing approaches, because methods represent a more appropriate level of reasoning on which to communicate;

for example, explanations can be readily provided from method preconditions, as explained in the next section.

8.2.5 Perspicuity

Because I use tactics, specified by methods, which mirror the user view of the problem, this makes the plans readable, and the skeleton configurations provided at the planning stage provide enough information for the user to make an informed choice as to whether the plan is worth executing. This is not necessarily true of, say, constraint based systems which work at a low level, as the reasons for the choices made may not be apparent on the surface.

Although I have not implemented any explanation facilities, this being beyond the scope of the project, tracing the plan will show which methods are being tried: it is also possible to examine these at the level of the preconditions. Thus we can see where a method fails to be applicable.

It is envisaged that more should be made of the explanatory possibilities of proof planning techniques in the future. This is discussed in the next section.

8.3 Possible extensions to CLEM

8.3.1 Immediate extensions

The immediate extensions which could be made are:

1. Including more processors, and more heuristics for choosing processors.

2. Including a strategy to provide the cheapest possible configuration. This would be achieved by suppressing the use of efficiency heuristics. It would be of limited use, though, as suggested in Chapter 3.

3. Including more knowledge linking devices with applications. At present, CLEM possesses only rudimentary knowledge about which terminals are suitable for which applications. This could be extended to printers, for instance. We would need a hierarchy of applications and devices to cope with multi-purpose devices: for example, if a terminal is to be used for data entry alone a 2392*A* terminal will be sufficient, but word processing requires a PC; the 2932*A* will not support word processing but the PC will support data entry as well, and it can be used for both applications.

4. More fundamentally, information does exist on the disk storage requirements of various applications, and if this could be incorporated there would be no need for the user to provide any explicit information about storage at all. Thus we could include a method for calculating the amount of disk capacity needed from applications, numbers of users, *etc.*

8.3.2 Further extensions

The extensions above all involve merely accumulating or incorporating new expert domain knowledge. More fundamental extensions which are desirable have already been touched on in Chapter 7, when discusing the shortcomings of CLEM.

The main problem seems to be that the introduction of supermethods is a double-edged sword. In situations where it is possible (and required) to find a configuration obeying all heuristic constraints, the use of the basic configuration supermethod is beneficial, in that significantly different solutions can be found quickly. However, if this is not the case, what we would ideally like is to find a solution by trying a constraint relation strategy, and maybe a second solution by trying a different relaxation strategy.

This is not possible within the current format, of course, as each supermethod (embodying a strategy) can succeed in many different ways.

What we really seem to be asking for is either

1. some way of dynamically introducing methods, or deleting them, and of allowing the user some choice over which is used, similar to the Hints Mechanism developed for CLAM (Negrete, 1991); or

2. a way of transforming solutions: taking one solution (which overloads a card cage, say) and transforming it to a solution which does not (but may instead overload a channel), analogous to the ideas of proof transformation given in Madden (1991).

Another important extension is the provision of explanation facilities. This is of vital importance if the user is to take any dynamic part in proceedings, as intimated above. The preconditions of methods make good explanation facilities a realistic possibility. This issue was discussed in Chapter 7 in connection with unrealizable specifications. Another use would be in providing a "running commentary" of the configuration process, which could be useful in system development or in teaching configuration skills.

8.4 Proof planning for IKBS

The motivation for turning AI in general, and the pursuit of IKBS in particular, into an engineering science is so that the resulting expert systems are *reliable*. Logic provides a firm basis on which to achieve this; in addition, meta-level reasoning, as captured by the proof planning methodology, gives us the flexibility to use this logic purposefully.

The next stage in the story of the "proof planning for IKBS" story would be to try other IKBS problems. I make no claim to have solved, or even have the basis of a solution for, all configuration, let alone all design, problems (Chapter 2, Section 2.4). My aim was only to build a system capable of configuring one group of hardware systems, and I felt I was fully stretched in developing a configuration theory which could deal with objects of such different forms.

It remains pertinent to ask whether this general *approach* is transportable to other domains, however. It was the question I asked at the beginning, whether we could transport proof planning from mathematics to configuration. Will a transfer to yet another domain be possible? Also, will the next domain tackled prove easier by virtue of the general expertise gained from the experience of tackling the configuration task? To answer this, we need to ask whether we have learned anything fundamental from this exercise. In particular, we need to ask whether there was something peculiar to configuration which made it possible.

For the configuration task, the salient features which made proof planning both possible and productive were:

1. An object-level theory was discernible, if a little difficult to formalize.

2. A large search space: combinatorial explosion at the object-level made the use of the meta-level desirable, over and above merely paying back the overhead.

3. The ability to separate factual from heuristic and control knowledge.

4. Fairly well-developed strategies for performing the task.

5. The likelihood that new products and new procedures would fit into the overall framework.

For more static domains, of course, the last of these is not an issue. In general, these features are common to many tasks involving reasoning.

The outcome of this study has been:

- The formalization of an initially very informal domain, where it was at first difficult to ascertain which category various bits of knowledge belonged to.

- The characterization of the configuration task as synthesis in the *proofs as objects* paradigm.

- The development of a theory of configuration, and of various strategies to assist with the task of configuring a hardware system to meet a specification.

- The conceptual division of the task into two views: the user view and the engineering view, of which the former is encapsulated by the preconditions and effects of methods, which are specified by tactics, and the latter is guaranteed correct by the execution of these tactics.

- The embodiment of strategies as proof plans.

- The implementation of CLEM, an automatic configurer capable of testing these ideas and developing new ones.

I would predict that many other domains could be tackled in this way. In particular:

1. Other kinds of configuration, as defined in Chapter 3, by following the same methodology, *viz.* using logic to encode an object-level theory for the domain, determining procedures and strategies and expressing these as proof plans.

2. Other design problems, for example in building. The advantage in this domain is in the existence of professional bodies with an interest in advancing common knowledge of how these tasks are performed, as opposed to the field of hardware configuration, where systems are developed by competing firms.

3. Using proof plans for playing bridge (Frank *et al*, 1992). This work has already begun.

8.5 Envoi

As Kuhn (1970) states, it is not enough, when promoting a new paradigm, to demolish the old one. Part of the attraction of a paradigm, once it becomes the "established" one, is that it leaves plenty of "holes". In science, this "hole-filling" activity can be seen as virtually all of what Kuhn calls "normal scientific activity". There is a good parallel in the field of expert systems — whether seen as "scientific activity" or not — in that maintaining XSEL, for example, is seen as a satisfying activity by the maintenance programmers who perform this task. In fact, it seems to be generally expected that rule-based systems will naturally require this kind of maintenance, and that the resulting problems of control will be dealt with cleverly in an implicit way if necessary. In the world of the computer hacker, the gradual tinkering away at flawed theories in Popper's ideal world can become a grotesque sacrifice of clarity on the altar of performance.

According to Kuhn's view of change, only gradually can a new approach be expected to find adherents, until, possibly, it becomes the new paradigm. I would expect a great deal of work to be necessary, in developing logically-based knowledge based systems incorporating meta-inference in a principled and explicit manner, for different problems and different domains.

However, I believe that such work is worth doing. Basic research in IKBS is needed to place the methodology for developing such systems on a firm logical basis if it is to be as dependable in the future as other engineering disciplines already are (Bundy, 1987a).

Most interesting tasks involve a mixture of different kinds of knowledge: factual, heuristic, and "knowing how" control knowledge. Most knowledge that we possess can be turned to several uses. The proof planning approach grasps these two facets of human understanding and can use them to good purpose.

References

Alden, Jeremy and Morgan, Robert. (1974). *Regional Planning: A Comprehensive View.* Leonard Hill Books.

Bachant, Judith and Soloway, Elliot. (March 1989). The engineering of XCON. *Communications of the ACM*, 32(3):311–317.

Ball, Michael. (1979). Cost-benefit analysis: a critique. In Green, Francis and Nore, Petter, (eds.), *Issues in political economy: a critical approach*, chapter 3, pages 63–88. The Macmillan Press Ltd, London.

Barker, Virginia E. and O'Connor, Dennis E. (March 1989). Expert systems for configuration at digital: XCON and beyond. *Communications of the ACM*, 32(3):298–310.

Bates, Joseph L. and Constable, Robert L. (January 1985). Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136.

Brachman, Ronald J. (1983). What IS-A Is and Isn't. *Computer*, 16(10):30—36.

Bundy, A. and Sterling, L.S. (September 1981). Meta-level inference in algebra. Research Paper 164, Dept. of Artificial Intelligence, Edinburgh, Presented at the workshop on logic programming for intelligent systems, Los Angeles, 1981. A revised version is available as Research Paper 273, Dept. Of Artificial Intelligence, Edinburgh.

Bundy, A. (1987a). AI bridges and dreams. Research Paper 300, Dept. of Artificial Intelligence, Edinburgh, Also in AI and Society, vol. 1 no. 1.

Bundy, A. (1987b). How to improve the reliability of expert systems. In Moralee, S., (ed.), *Research and Development in Expert Systems IV*, pages 3–17. C.U.P. Also available as Research Paper 336, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In *9th Conference on Automated Deduction*, pages 111–120. Longer version available as Research Paper 349, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A. (1991). A science of reasoning. In Lassez, J-L. and Plotkin, G., (eds.), *Computational Logic: Essays in Honor of Alan Robinson.* MIT Press. Also available as Research Paper 445, Dept. Of Artificial Intelligence, Edinburgh.

Bundy, A., van Harmelen, F., Hesketh, J., Smaill, A. and Stevens, A. (1989). A rational reconstruction and extension of recursion analysis. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufman. Available as Research Paper 419, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A., Smaill, A. and Hesketh, J. (1990a). Turning eureka steps into calculations in automatic program synthesis. In Clarke, S.L.H., (ed.), *Proceedings of UK IT 90*, pages 221–6. Also available as Research Paper 448, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A., Smaill, A. and Wiggins, G. (1990b). The synthesis of logic programs from inductive proofs. In Lloyd, J., (ed.), *Computational Logic*, pages 135–149. Springer-Verlag. Esprit Basic Research Series. Also available as Research Paper 501, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1991a). Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, To appear in Artificial Intelligence.

Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991b). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available as Research Paper 413, Dept. Of Artificial Intelligence, Edinburgh.

Chandrasekaran, B., Josephson, J. and Herman, D. (1987). The generic task toolset: High level languages for the construction of planning and problem solving

systems. Technical Report 87-BC-TOOLEV, OSU, Presented at the Workshop on Space Telerobotics, Pasadena CA.

Constable, R.L., Allen, S.F., Bromley, H.M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

Frank, I., Basin, D. A. and Bundy, A. (1992). An adaptation of proof-planning to Declarer play in Bridge. Research Paper 575, Dept. of Artificial Intelligence, Edinburgh, Shorter version appears in the proceedings of ECAI-92.

Frayman, F. and Mittal, S. (1987). Cossack: a constraints-based expert system for configuration tasks. In *Proceedings of the 2nd International Conference on Applications of AI to Engineering*, Boston MA.

Freeman, Michael W., (December 1985). Case study of the beacon project. IEEE Videoconference seminar on Expert Systems and Prolog.

Hall, Peter. (1975). *Urban and Regional Planning.* Penguin Books Ltd, Harmondsworth.

Harmon, P. (1987). Digital equipment corp saves millions with expert systems! *Expert Systems Strategies*, 3(8):1–13.

Hayes, P. (1977). In defence of logic. In *Proceedings of IJCAI-77.* International Joint Conference on Artificial Intelligence.

Hesketh, J.T. (1991). *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* Unpublished Ph.D. thesis, University of Edinburgh.

Horn, C. (1988). The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, Edinburgh, The Edinburgh version of Nurprl has been renamed Oyster.

Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov, A., (ed.), *International Conference on Logic Programming and*

Automated Reasoning – LPAR 92, St. Petersburg, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag.

Jackson, P. (1986). *Introduction to Expert Systems.* Addison-Wesley.

Kuhn, Thomas. (1970). *The structure of scientific revolutions.* University of Chicago Press, second edition.

Laird, John E., Newell, Allen and Rosenbloom, Paul S. (1987). Soar: an architecture for general intelligence. *Artificial Intelligence*, 33.

Lowe, Helen. (1988). Empirical evaluation of meta-level interpreters. Unpublished M.Sc. thesis, University of Edinburgh.

Lowe, Helen. (1991a). Extending the proof plan methodology to computer configuration problems. *Artificial Intelligence Applications Journal*, 5(3). Also available from Edinburgh as Research Paper 537.

Lowe, Helen. (1991b). The use of theorem proving techniques in expert systems for configuration. In Rault, J.-C., (ed.), *Proceedings of the Eleventh International Workshop on Expert Systems and their Applications, Avignon.* EC2. Also available from Edinburgh as Research Paper 536.

Lowe, Helen. (1993). The CLEM configuration system, user manual and programmer manual. Technical Paper (forthcoming), Dept. of Artificial Intelligence, Edinburgh.

Madden, P. (1991). *Automated Program Transformation Through Proof Transformation.* Unpublished Ph.D. thesis, University of Edinburgh.

Manning, Alistair. (1992). Representing preference in proof plans. Unpublished M.Sc. thesis, DAI.

Martin-Löf, Per. (August 1979). Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy*

of Science, pages 153–175, Hanover. Published by North Holland, Amsterdam. 1982.

McDermott, John. (1982). R1: a rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88.

Merry, Martin. (1992). Rubicon — a computer configuration system. In *Expert Systems 1992.*

Mittal, Sanjay and Frayman, Felix. (1989). Towards a generic model of configuration tasks. In *Proceedings of IJCAI-89*, pages 1395–1401. IJCAI-89.

Mumford, Enid and Macdonald, W.B. (1989). *XSEL's progress: the continuing journey of an expert system.* Chichester Wiley.

Negrete, Santiago. (1991). Proof plans with hints. Unpublished M.Sc. thesis, University of Edinburgh.

Owen, S.G. (1988). The development of explicit interpreters and transformers to control reasoning about protein topology. Technical report, Hewlett-Packard Research Laboratories, Bristol.

Pierick, Jeff. (1986). A knowledge representation system for systems dealing with hardware configuration. In Kehler, T. and Rosenschein, S., (eds.), *Proceedings of the Fifth National Conf on Artificial Intelligence*, pages 991 – 995, Philadelphia. AAAI.

Polit, S. (December 1985). R1 and beyond. *The AI Magazine.*

Polya, G. (1945). *How to solve it.* Princeton University Press.

Popper, Karl R. (1961). *The Poverty of Historicism.* Routledge & Kegan Paul, London, (Revised version with some corrections.).

Popper, Karl R. (1989). *Conjectures and Refutations.* Routledge & Kegan Paul, London.

Popper, Karl R. (1990). *The Logic of Scientific Discovery.* Unwin Hyman.

Rosenbloom, Paul S., Laird, John E., McDermott, John, Newell, Allen and Orciuch, Edmund. (September 1985). R1-soar: an experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and machine intelligence*, PAMI-7(5):561–568.

Searls, David B. and Norton, Lewis M. (1990). Logic-based configuration with a semantic network. *Journal of Logic Programming*, 1:53–73.

Silver, B. (1985). *Meta-level inference: representing and learning control information in Artificial Intelligence.* North Holland, Revised version of the author's PhD thesis.

Smith, Adam. (1776). *An inquiry into the nature and causes of the wealth of nations.* Penguin Books Ltd, Harmondsworth.

Soloway, E., Bachant, J. and Jensen, K. (July 1987). Assessing the maintainability of XCON-in-RIME: coping with the problems of a very large rule-base. In *6th National Conference on Artificial Intelligence*, Philadelphia. Sponsered by the American Association for Artificial Intelligence.

Stefik, Mark. (1981). Molgen part 1: Planning with constraints. *Artificial Intelligence*, 16:111–139.

van Harmelen, F. (1989). The CLAM proof planner, user manual and programmer manual. Technical Paper 4, Dept. of Artificial Intelligence, Edinburgh.

Wielinger, B.J., Schreiber, A.Th. and Breuker, J.A. (1992). Kads: a modelling approach to knowledge engineering. *Knowledge Acquisition*, 4:5–53.

Wiggins, G. A. (1992). Synthesis and transformation of logic programs in the Whelk proof development system. In Apt, K. R., (ed.), *Proceedings of JICSLP-92.*

Wu, Harry, Chun, Hon Wai and Mimo, A. (1986). Iscs - a tool for constructing knowledge based system configurators. In Kehler, T. and Rosenschein, S., (eds.), *Proceedings of the Fifth National Conf on Artificial Intelligence*, pages 1015 – 1021, Philadelphia. AAAI.

Appendix A

Definitions

Utility (Definition 3.1)

The *utility* of a system is a multi-dimensional measure of its worth to the user. These dimensions may vary, but at present we enumerate them as

(1) cheapness (inverse of cost)

(2) efficiency

(3) expandability

(4) anti-obsolescence (inverse of obsolescence)

(5) technological innovation

Locally optimal solution (Definition 3.2)

A *locally optimal solution* is one which cannot be increased (improved) along one dimension without a simultaneous decrease (deterioration) along at least one other dimension.

Locally sub-optimal solution

A *locally sub-optimal solution* is one which can be increased (improved) along one dimension without a simultaneous decrease (deterioration) along any other dimension.

Globally optimal solution

A *globally optimal solution* is one which cannot increased (improved) along any dimension.

Usage (Definition 4.1)

The *usage* of a component (of computer hardware), or group of components, is its function, in terms of the service it provides to the user of the computer configuration. A component may have more than one usage.

Value (Definition 4.2)

The *value* of a component, or group of components, is the set of usages, paired where appropriate with a measure of how well each function is performed. This measure may be quantitative or qualitative.

Valid connection (Definition 4.3)

Definition A.1 *A component, comp, has a* valid *connection if the function connected_via(comp, c) is defined and returns a value, i.e.*

$$\exists \tau \exists x{:}\tau.connected\_via(comp, c) = x$$

Legal configuration (Definition 4.4)

A configuration c is *legal* if

(i) There are valid instantiations for the values of all functions of the form *essential_component(c)*, where *essential_component* stands in turn for each of the members of a designated set of essential parts.

(ii) Every component object of c has a valid connection.

Appendix B

Glossary of Terms

B.1 Glossary of Computer Terms

Access speed: the speed at which data can be obtained from a storage device: *e.g.* from a disk drive.

Active users: of those users who are currently logged on to a system, those actually involved in processing, as opposed to those users who are logged on, but idle.

Backup (storage/space): storage space used to back up (important) data. Often provided on tape drives.

Cable: various kinds of cables exist; we have examples of RS-232 and RS-422 cables.

Card cage: a frame holding the **central processor**, **memory boards**, and **interface cards**.

Central processor: contains memory, control unti, and arithmetic unit.

Channel: the path along which data is transferred.

Connector: a component which does not itself provide utility to the user (*c.f.* **device**) but is necessary in order to configure components which do. I have used this term to encompass **card cage**, **cable**, and various **interfaces**.

Console: used for communication between the computer operator and the system, consisting of a VDU and a keyboard.

Device: a peripheral unit, providing some utility.

Disk drive: A storage device. The *capacity* of a disk drive is usually expressed in megabytes (Mb) or gigabytes (Gb).

Fibre-optic link: A means of connecting a device which enables very fast data transfer.

Fibre-optic disk drive: A disk drive connected by means of a fibre-optic link.

GIC: general interface channel, used for connecting system devices (similar to the HPIB). Used by Series 70 systems, but not by HP 950 systems.

HPIB: used for connecting system devices *via* HPIB cables.

HPFL: used for connecting devices *via* fibre optic links.

Interface: boundary between the central system and a device in a configuration. **Interface cards** occupy slots in a card cage; those found in HP 950 systems are MUXes, LANICs, HPIB, and HPFL cards.

LANIC:local area network interface card. Also used, in the HP 950 systems, for configuring serial devices: terminals and some printers.

Memory board: a board, usually particular to a given processor, providing a fixed amount of internal memory capacity (usually measured in megabytes).

Modem: A component allowing data to be transmitted over telephone lines.

Multiplexor (MUX): a device allowing the processor to be connected to a number of communication channels.

PC: see **Personal Computer**

Personal Computer: A desk top computer, taken here to incorporate a VDU, a keyboard, memory, and some means of storing data. It can be connected to a mainframe system or to a network of other computers.

Port: A single data channel whereby a device may be connected.

Printer: An output device. Many different kinds exist, *e.g.* page printers, line printers, dot matrix printers, laser printers.

Screen: VDU; or (*coll.*) terminal, PC

Serial device: A device which is connected *via* a RS-232 or RS-422 cable, or a modem, to a serial port.

Storage: means of holding data: *e.g.* disks, tapes.

System device: A device which is connected *via* an HPIB or HPFL cable, or *via* a GIC

System disk (storage/space): A disk drive which is to be used to store system software, *i.e.* software to enable the system to function for the benefit of the user, *e.g.* the operating system *c.f.* **User disk**.

Tape drive: A storage device. **Access speed** is slower than for disk drives. Nowadays often used for **backup** and data transfer.

Terminal: A device consisting of a VDU and a keyboard, for providing output and input respectively to a computer or network.

User disk (storage/space): A disk drive which is to be used to store the data and programs written by or for to the users of the computer system, *e.g.* applications software. *c.f.* **System disk**.

B.2 Glossary of Types and Tactics

Examples of these may be found in Appendix D.1.1.

B.2.1 Simple types

`cardcage`	`printer`
`channel`	`processor`
`disk`	`rs232_portgroup`
`fchannel`	`rs232_serial_cable`
`fdisk`	`rs422_portgroup`

```
flink                    rs422_serial_cable
lanic                    serial_conn
memory                   system_cable
modem_portgroup          tape
mux                      terminal
```

B.2.2 Complex types

```
list(Type)
cross(device,serial_cable)
cross(device,system_cable)
cross(fdisk,flink)
cross(cross(device,cable),conn)
cross(cardcage,(list(mux)<>list(lanic)<>list(channel)<>list(fchannel)))
mux_term = cross(lanic,list(terminal))
lanic_term = cross(lanic,list(dtc_term))
dtc_term = cross(serial_conn,list(portgroup))
portgroup_term = cross(portgroup,list(serial_device))
channel_term = cross(channel,(list(disk)<>list(tape)<>list(printer)))
fchannel_term = cross(fchannel,list(fdisk)
```

B.2.3 Supertypes

```
cable = serial_cable + system_cable
device = sdisk + tape + serial_device
interface = mux + lanic + channel + fchannel
portgroup = rs232_portgroup + rs422_portgroup + modem_portgroup
sdisk = disk + fdisk
serial_cable = rs232_serial_cable + rs422_serial_cable
serial_device = terminal + printer
```

B.2.4 Tactics

These are documented in Appendix D.2.1

```
add_backup(-Config,-Tapes)
add_system_disks(-Config,-System_disks)
configure_cc(-Config,+Cardcage)
configure_device(-Config,+Device,?Interface)
configure_device_list(-Config,?Device_list)
configure_ic(-Config,+Channel,?Cardcage)
configure_portgroups(-Config,+Portgroup,?DTC)
configure_processor(-Config,+Processor)
configure_serial_connection(-Config,+DTC,?LANIC)
configure_serial_device(-Config,+Device,?Interface)
disk_storage(-Config,+Capacity,?Disks)
h_configure_device(-Config,+Device,?Interface)
h_configure_ic(-Config,+Channel,?Cardcage)
is_legal(?Config)
match_attributes(+Goal,?Device)
special(-Config)
tape_capacity(-Config,+Capacity,?Tapes)
total_disk_storage(-Config,+Capacity,+System_disks,?Disks)
```

Appendix C

CLEM Manual

The CLEM Configuration System

User Manual

and

Programmer Manual

Notation

1. Normal text is in normal roman font.
2. New terms are introduced in *italic roman font.*
3. Prolog code is in `typewriter font`.
4. Predicates will be denoted by `f/n`, which stands for the Prolog predicate `f` of arity n.

Format

1. Predicate definitions are headed by the name of the predicate surrounded by horizontal lines.
2. There are separate predicate and normal (keyword) indexes.
3. The defining entry for a predicate is distinguished in the predicate index by an underlined page number.

In the specification of a predicate, variables are annotated with mode annotations: `+`, `-` or `?`. This notation is borrowed from the Quintus Reference Manual.

+ This argument is an input to the predicate. It must initially be instantiated or the predicate fails (or behaves unpredictably).

- This argument is an output. It is returned by the predicate. That is, the output value is unified with any value which was supplied for this argument. The predicate fails if this unification fails.

? The argument may be either input or output, and may be instantiated or not, as required by its application. I often use this when I expect an argument to be *partially instantiated*, *i.e.* some of its subterms to be ground and others not.

Note that if a mode is not given for a predicate, it does not mean that it it can never be used in that mode. It means simply that I do not expect the predicate ever to be used in that mode, and, if it were, I could not guarantee it.

C.1 Introduction

C.1.1 CLEM: an expert configurer

CLEM is the implementation in Prolog of a proof planning system for configuring computer hardware to meet specifications. Like CLAM (van Harmelen, 1989), it uses *methods*, which are specifications of *tactics*, to build *proof plans*. When these plans are executed, the tactic, which is a piece of Prolog code, corresponding to each method is run. Each tactic carries out part of the task of synthesizing a hardware configuration. After all the tactics have been executed, we have a full, legal configuration which meets the specification.

Typically, specifications mention those aspects of the configuration which are of interest to the user, or customer, of the hardware system: applications, terminals, printers, *etc.* They may well not mention other, vital components, such as processor, backing storage, *etc.* The type specification of configurations ensures that only well-formed objects can be formed, thus generating all these vital components and ensuring that all components are correctly connected.

The search space is potentially very large, and CLEM incorporates a few design heuristics in order to try and generate "good" solutions in real time. This is discussed in Section C.6.3 (pg.228).

C.1.2 Structure of this manual

In this manual you will find the following information:

- An introduction to the domain of configuration, in Section C.2 (pg.205).

- The knowledge base of the CLEM system: *i.e.* the object-level knowledge appertaining to components and their attributes and the rules of configuration, in Section C.3 (pg.207).
- Heuristic knowledge used in configuration, in Section C.4 (pg.217).
- A note on maintaining the object-level knowledge, in Section C.5 (pg.218).
- The mechanism for representing methods and the language that can be used for formulating them, the methods that are currently implemented in CLEM, and the tactics that can be used to execute proof plans, in Section C.6 (pg.219).
- The representation of configuration schemes, in Section C.6.3 (pg.228).
- The planner used to build proof plans out of these methods, in Section C.7 (pg.235).
- Utilities (pretty-printer, tracer, statistics collection), in Section C.8 (pg.236).
- Section C.9 (pg.239) describes how to get started with CLEM.
- Section C.10 (pg.241) describes the organization of CLEM's source code.

C.2 The configuration domain

- 70 screens for data entry
- 5 screens for program development
- 20 screens for electronic mail and spreadsheet application
- 1 line printer with a speed of at least 900 lpm
- 2 laser printers, each with a speed of at least 45 ppm
- 1 laser printer with double-sided printing option
- approximately 1.7 GBytes of mass storage

Figure C–1: Invitation to tender specifications

Customers for computer hardware generally think in terms of the use to which the system is to be put, and they will go through the process of discussing customer needs with the sales representative for the computer firm. Issues include the applications to be run, the number of users involved, and possibly other considerations such as cost, and the future needs of the company as regards expanding

```
HERE IS THE SYNTHESIZED CONFIGURATION
-------------------------------------
processor:     950(2)
system disks:
     7937H(12) with cable hpib-cable(15)
user disks:
     7937H(2) with cable hpib-cable(4)
     7937H(3) with cable hpib-cable(5)
          ...   ...   ...   ...   ...

     7937H(11) with cable hpib-cable(13)
tape drives:
     7980A(2) with cable hpib-cable(14)
printers:
     2564A(2) with cable hpib-cable(2)
     2235A(2) with cable hpib-cable(3)
terminals:
     2392A(3) with cable 40242X(2)
     2392A(4) with cable 40242X(4)
          ...   ...   ...   ...   ...

     2392A(18) with cable 40242X(18)
channel interfaces:
     hpib(2)          with devices
                                  7937H(2) 7937H(3) 2564A(2) 2235A(2)
     hpib(3)          with devices
                                  7937H(4) 7937H(5) 7937H(6) 7980A(2)
     hpib(4)          with devices
                                  7937H(7) 7937H(8) 7937H(9)
     hpib(5)          with devices
                                  7937H(10) 7937H(11) 7937H(12)
serial connections:
     lnc(2) with DTCs:
        dtc(3) connecting port groups:
            rs232ports(3)         rs232ports(4)

Devices connected via serial ports as follows:
rs232ports(3)
     2392A(3)        ...
rs232ports(4)
     2392A(11)       ...
card cages:
     950-cardcage(11)  with interfaces
                                  mx(2) mx(3) lnc(2) hpib(2) hpib(4)
     950-cardcage(12)  with interfaces
                                  hpib(3)
     950-cardcage(13)  with interfaces
                                  hpib(5)
yes
| ?-
```

Figure C–2: An example configuration output from CLEM (abbreviated)

or upgrading any hardware configured. This will lead, either formally to an *invitation to tender*, where the customer presents the requirements in written form and computer firms are invited to submit tenders; or informally to a request that the sales representative come up with suggestions. In either case, representatives of the computer hardware firm will detail the hardware to be configured, with itemized costs, and indications of any parts of the configuration which either fall below or exceed the specification, with reasons.

A typical (but fictitious) specification drawn up at the invitation to tender stage is shown in Figure C–1 (pg.205). A possible configuration meeting this specification can be seen in Figure C–2 (pg.206).

Note that the examples given are of *large-scale* computer systems. The problems involved with smaller systems are less acute, involving more constrained search. The large systems are a better test of the proof planning ideas. The knowledge base is taken from information on HP Series 3000 systems; however, this represents a "snapshot" of information which is constantly changing, and no guarantee is given as to its current applicability. In any case, only a sample is used, enough to demonstrate the ability of the system to deal with "scaled-up" problems.

However, it should be possible, using this manual, to update and scale up CLEM's knowledge base so that it can deal with realistic, and up-to-date, data. See especially Section C.5 (pg.218) for details of this.

C.3 Object-level knowledge

C.3.1 Types of objects

The fundamental unit is the *component*, and every component has a *type*. Components may be connected to each other, and the compound object so formed also has a (compound) type.

A Prolog file `types.pl` gives the set of components currently recorded in the knowledge base. It contains two predicates, `type/2`, and `supertype/2`.

• `type(?Component(Index),?Type)`

This predicate can be used in modes:

1. `type(+Component(Index),-Type)`

   to find the *Type* of a given *Component*.

2. `type(-Component(Index),+Type)`

   to find (*i.e.* generate) an indexed *Component* of type *Type*.

3. `type(+Component(Index),+Type)`

   to verify that *Component(Index)* is of type *Type*.

4. `type(-Component(Index),-Type)`

   will generate, successively, an example of each component held in the knowledge base. This is of limited usefulness, except for periodically checking against hardware lists for maintenance purposes.

The current set of components includes two processors, various memory modules, card cages, interfaces, and connecting components, eight different models of type `disk`, three of type `tape`, sixteen printers, and four terminals. A fragment of the file is shown in Figure C–3 (pg.208). □

```
%%% These clauses are all of the form
%%% type(comp-name(index),comp-type):-nat(comp-name,index).
%%% For example: type('950'(N),processor)
%%%  holds provided N is a natural number.

%%% note the function of the predicate
%%%         nat(+component,?index)
%%%  if index is not instantiated, then a unique
%%%  index is created for this component type.
%%% Thus is can be used to check, or generate, a uniquely
%%% referenced component.

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %                                %
        %     disk drives                %
        %                                %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        type('7937H'(N),disk):- nat('7937H',N).
        type('7937XP'(N),disk):- nat('7937XP',N).
        type('7933H'(N),disk):- nat('7933H',N).
        type('7933XP'(N),disk):- nat('7933XP',N).
        type('7935H'(N),disk):- nat('7935H',N).
        type('7935XP'(N),disk):- nat('7935XP',N).
        type('7936H'(N),disk):- nat('7936H',N).
        type('7936XP'(N),disk):- nat('7936XP',N).
```

Figure C–3: Segment of file `types.pl`

If the predicate `type/2` is called in mode 2 above, for example in order to generate a component of type `disk`, as shown in Figure C–4 (pg.209), then all possible models of disk can be generated.

`type(?List,list(?Type))` will either check that `List` is a list of objects, all of which are of type `Type`, or, alternatively, it will generate a list of objects of a required type.

There are various compound types: for example

```
| ?- type(Component,disk).

Component = '7937H'(13) ;

Component = '7937XP'(2) ;

Component = '7933H'(2) ;

Component = '7933XP'(2) ;

Component = '7935H'(2) ;

Component = '7935XP'(2) ;

Component = '7936H'(2) ;

Component = '7936XP'(2) ;

no
| ?-
```

Figure C–4: Calling `type/2`

```
%%% Disks, tapes, and printers can be connected via system cables.
type([Dev,Cable],cross(device,system_cable)):-
                (type(Dev,disk);type(Dev,tape);type(Dev,printer)),
                type(Cable,system_cable).
```

represents the rule which states that a disk, tape, or printer connected to a component of type `system_cable` is a valid compound object.

- **`supertype(?Component,?Supertype)`**

The predicate `supertype/2` is useful in a few cases, mostly for ease in dealing with polymorphic functions, and where we want to allow lists of objects of different types. We have various supertypes over different kinds of device (serial and system devices), over different types of disks, portgroups, interfaces and cables. Lastly, the list supertype is for lists of objects all of the same supertype (not necessarily of the same type). □

C.3.2 Attributes of objects

Depending on their type, objects (components) possess *attributes*. For example, a disk drive will have a capacity, in megabytes, representing the amount of data which can be stored on it; a printer will have a speed, expressed either in characters

per second, or lines or pages per minute. The attributes of components given in `types.pl` can be found in the file `config-attr.pl`.

C.3.3 Attributes of partial configurations

Objects grouped together may also possess attributes; for example, just as we have the capacity of a single drive we may want the (total) capacity of all the disk drives in a configuration.

In addition, there are certain other predicates giving properties of a configuration which cannot simply be checked by examining the property of an individual component: this mostly involves inequality reasoning. These can also be found in the file `config-attr.pl`.

• `capacity(?Component:Type,?Capacity)`

This can be called in modes:

1. `capacity(+Component,-Capacity)`

   to discover the capacity of a component (disk, tape) in megabytes.

2. `capacity(-Component,-Capacity)`

   to generate components possessing an attribute `capacity`; these will be type-checked and used as shown next.

This predicate can be applied to various types of component; so far we have only examples of disk drives and tape drives. □

• `capacity(?Component:list(Type),+Capacity)`

This predicate is shown in Figure C–5 (pg.211).

Note that *no heuristics* are incorporated in this object-level knowledge. Thus, for instance:

- It is quite feasible to mix different kinds of disk drive in one configuration.
- It is possible to generate partial configurations *well over* the desired capacity.

Note that in both cases of `capacity/2`, the responsibility for checking that the device(s) generated are of the correct type lie elsewhere. □

• `printer_type(?Printer,?Ptype)`

This is used in two modes:

```
capacity([],0).
capacity([H|T],Cap):-
    capacity(H,C1),
    capacity(T,C2),
    Cap =< C1 + C2.
```

Figure C–5: Capacity of partial configurations

1. `printer_type(+Printer,-Ptype)`

   to discover whether a particular printer is a line printer, a laser printer, a page printer ,*etc.*

2. `printer_type(-Printer,+Ptype)`

   to generate a printer of the required type.

It is possible (though I have no examples) that a printer can be of more than one `printer_type`. □

• speed(?Printer,+Units,?Speed)

This is called in either mode:

1. `speed(+Printer,+Units,-Speed)`

   to find the speed of a printer expressed in `Units`, *i.e.* lpm (lines per minute), cps (characters per second), ppm (pages per minute) or vpm (volume per month).

2. `speed(-Printer,+Units,+Speed)`

   to generate a printer with the required speed.

□

• lpm(?Device,?Speed)

This is used in either mode

1. `lpm(+Device,-Speed)`

   to discover whether `Device` has *at least* speed `Speed`.

2. `lpm(-Device,+Speed)`

   to generate a `Device` with *at least* a given `Speed`.

□

• `cps(?Device,?Speed)`

As `lpm/2` above, only for speed measured in cps. □

• `ppm(?Device,?Speed)`

As `lpm/2` above, only for speed measured in ppm. □

• `throughput(?Device,?Vol)`

This is used in either mode

1. `throughput(+Device,-Vol)`

   to discover whether `Device` can cope with *at least* `Vol` pages per month.

2. `throughput(-Device,+Vol)`

   to generate a `Device` which can cope with *at least* a given number of pages per month.

□

• `supports_application(?Application,+DeviceType,?Device)`

This is used in modes

1. `supports_application(+Application,+DeviceType,?Device)`

   to generate a `Device` of type `DeviceType` (terminal, printer) which will support a given `Application` (spreadsheet, data entry, program development, word processing), or to check that a given device will do so.

2. `supports_application(-Application,+DeviceType,+Device)`

   will find which applications a given device is thought suitable for.

□

• `has_graphics(?Terminal)`

Used either to check that a given `Terminal` has facilities for displaying graphics, or to generate one which does. □

• `modem_suitable(?Printer)`

Use either to check that a given `Printer` can be connected via a modem (not all printers can), or generates one which can be so connected. □

```
disk_capacity(Config,Cap):-
    disks(Config,Dks),
    capacity(Dks,Cap).
```

Figure C–6: Disk capacity of a configuration

- `os_suitable(+Processor,-DiskList)`

This finds a list of disk drives which, together, is suitable for holding the operating system for a given processor. There will be several possibilities, in general.

Most probably there should be a third argument, for the version of the operating system. However, I have assumed that we want the 'default', latest version, and that this only is stored. It would be very simple to remove this simplification, but in the absence of much data, and since the scope of this project did not extend to software configuration, this was not done. □

- `disk_capacity(?Config,?Capacity)`

This predicate, shown in Figure C–6 (pg.213), returns the user disk capacity of a configuration.

It could be easily generalized to take account of configurations in which, depending on the processor used and devices supported, we can take account of storage devices other than disk drives for storing users' data. □

- `total_disk_capacity(?Config,?Capacity)`

This is like `disk_capacity` except that the total of all disk space, whether for system use or for user data, is returned. □

- `backup_capacity(?Config,?Capacity)`

Like `disk_capacity/2`, but for backup devices. □

C.3.4 Axioms for configuring hardware systems

Configurations are represented as tuples. In Prolog these are implemented as lists of elements in a fixed order. Projections are used to identify particular partial configurations; *e.g.* the disk drives of a configuration. Examples of these, for 950 processor systems, are shown in Figure C–7 (pg.214).

```
processors([P|_],P):-
    type(P,list(processor)).

system_disks([P,Sys|_],Sys):-
    os_suitable(P,Sys),
    supertype(Sys,list(sdisk)).

muxes([['950'(_)|_],_,_,_,_,_,_,_,_,Mx,_,_,_,_,_,_,_,_,_],Mx):-
    type(Mx,list(mux_term)).
```

Figure C–7: Projections for accessing parts of 950 configurations

Thus a configuration is represented in Prolog as a list. The first element of the list is always a list of processors. Actually, I have looked only at single-processor systems so this has been a singleton list hitherto, but there is scope to synthesize multi-processor systems, given the necessary domain knowledge.

The second item is always the devices used to store the operating system. For the systems I have looked at, this is one or more disk drives. There seem to be strict rules governing what can be used for each processor, so this is checked via the predicate `os_suitable` (*qv*).

The third example projection given in Figure C–7 (pg.214) is that projecting the MUXes for a '950' system. Note that the number, and types, of elements in the configuration list term is processor dependent. Many processors do not use certain types of components at all. 950 systems use MUXes, but only for configuring the console. Other systems use allow MUXES for configuring any kind of terminal.

The projections for 950 and 925 systems have been stored. Those for other processor systems can be added, as indicated in Section C.5 (pg.218).

We also need axioms to define what constitutes a *legal* system.

• `legal(?Config)`

This predicate is usually called on a partially instantiated configuration term `Config`. It calls, in turn, two predicates, `special/1` and `general/1`. If it succeeds, then the configuration is legal with the explicitly named (ground) components as they stand, and all uninstantiated terms and tails of list may be "closed off". □

• `special(?Config)`

This matches `Config` against one of the template configurations. For example, if `Config = [['950'(_)]|_]` then `Config` must be matched against a list of 17 elements, starting with `['950'(_)]`, and including system disks, MUXes, LANICs, *etc.* in the appropriate positions. □

• `general(?Config)`

This checks that certain components are present (and ground). The essential components are processor, storage for operating system, backup devices, and console. It also picks out all the devices and interfaces from the configuration and checks that they are connected. This is the same for all systems, regardless of processor. □

C.3.5 Connections

We need functions to express relationships between components; namely the fact that two components are *connected*, that one component acts as the connection for another, *etc.*

In the underlying logic, we use functions since connections are unique in this domain. However, in the Prolog implementation a lot of work is involved in ensuring this functional relationship holds, and a device is not allowed to have two cables, for instance. The rules concerning connection of components are to be found in `conn.pl`.

There are two predicates defined in this file.

• `connect_cable(+Device,?Cable,?Config)`

In general, `Config` is partially instantiated.

In mode

- `connect_cable(+Device,-Cable,?Config)`,

this predicate either discovers which cable belongs to a given device, or, if a cable has not yet been explicitly synthesized for that device, it generates one of the correct type, *e.g.*

```
** (5333) 8 Call: connect_cable('7937H'(1),_84277, ... )
** (5333) 8 Done: connect_cable('7937H'(1),'hpib-cable'(1), ... )
```

It can also (more rarely) be used in specifications in mode

- `connect_cable(+Device,+Cable,?Config)`,

to force a given cable on a device. However, it is more likely that the cable will be only partially instantiated (*i.e.* `'40242P(_)'`, where the variable _ will become instantiated to a unique index, *e.g.*

```
** (7536) 4 Call: connect_cable('2392A'(2),'40242P'(_7424), ... )
** (7536) 4 Exit: connect_cable('2392A'(2),'40242P'(4), ... )
```

Note that "cable" is a convenient catchall term encompassing not only "cables" in the accepted sense but also modem connections *etc.* □

• `connected_via(?Component1,?Component2,?Config)`

This can be used in modes as follows:

1. connected_via(+Component1,-Component2,?Config)

   To find how `Component1` is connected; or to generate a unique connection if it is not already connected.

2. connected_via(-Component1,+Component2,?Config)

   to discover what components are connected via a given (connecting) component.

`Component1` may of various types, *e.g.* disk, printer, MUX. Depending on what `Component1` is, then `Component2` may be of a number of types, *e.g.* channel, portgroup, cardcage.

If a connection is generated, then this is done by adding it to the unsubstantiated tail of the appropriate list term of `Config`, which is (in general) partially instantiated. □

C.3.6 Constraints

In the file `constraints.pl` are found all the *hard* constraints applicable to configuration. Obeying these constraints ensures that legal limits are not exceeded and that incompatible devices are not connected to the same place.

• `limit(+Component,+Type,-N)`

Given a component and a type of component, this predicate returns the number of components of that type (or supertype) which may be connected to the given component. For example:

```
| ?- limit(hpib(_),disk,N).

N = 4

| ?- limit(hpib(_),device,N).

N = 6
```

so any HPIB channel may have six devices connected to it, of which at most four may be disk drives. Note that `device` is a supertype. □

- `not_share_with(?Dev1,?Dev2,?Config)`

This is used to identify devices which may not share a connection. Examples of this are the `2680A` printer which fouls up any disk access it shares a channel with; and there is also a general rule that the system disk drive(s) and the system backup tape(s) should not share channels. □

C.4 Heuristic knowledge

Section C.3.6 (pg.216) refers to various *hard constraints* in the configuration domain. However, there are often corresponding *soft* constraints, and these may be found in the file `heuristics.pl`.

- `heur_limit(+Component,+Type,-N)`

This is the same as `limit/3` (*qv*) except that the soft, and not the hard limit is returned as `N`. *e.g.* (compare with the trace above):

```
| ?- heur_limit(hpib(_),disk,N).

N = 3

| ?- heur_limit(hpib(_),device,N).

N = 4
```

□

- `h_connected_via(?Component1,?Component2,?Config)`

This is like `connected_via/3` except that it discovers whether, or ensures that, a component is connected so as to obey *heuristic*, as opposed to *hard* constraints. □

C.5 Augmenting the knowledge base

The knowledge base was a "snapshot" of HP 3000 systems at a particular time and does not represent an unchanging body of knowledge by any means: indeed, knowledge bases for computer components are notoriously transitory! To this end, some guidance should be given here as to how to maintain the knowledge base. Ideally, this task would be partially automated: this was not my brief in this project, however. Nevertheless, a few simple guidelines should suffice.

- A single new component, of an existing type, is straightforward to add. For example, a tape drive is added to the current list of tape drives in `types.pl` as the entry

  ```
  type(<model-name>(_),tape).
  ```

  where `<model-name>` stands for the name (or, usually, number) assigned to the new model of tape drive; the blank argument is to hold an index, so that individual components of this model can be distinguished, and `tape` is the type of all such components.

- It is important to record all the attributes of the new component. Look in the files of attributes, constraints, and maybe heuristics, to see which attributes are appropriate. For example, for tapes you will see that the *capacity* is always recorded.

- For planning purposes, a `guess_type` entry will be required. See the file `preds.pl` for this. ultimately, this will be automated.

- If there are constraints involved, then enter these in the files `constraints.pl` (for hard constraints) and `heuristics.pl` (for soft constraints).

- If a new type is invented, then it can be added in the same way. However, there will be, in addition, various rules to do with connectivity. These may parallel other types. For example, a new type of storage component may be capable of being connected via HPIB cables so its connection rules will mirror that of tape and disk drives. See the file `conn.pl` for these rules.

If a "completely different" processor system is added, then appropriate axioms and templates must be given. Decide what device- and component-types are supported by the system. Draw up projections like the ones shown for the 950 systems in `axioms.pl`, and a template as given in `tactics.pl` (see Section C.6.1 (pg.219)).

```
configure_device(Device,Interface,Config):-
    connect_cable(Device,Cable,Config),
    connected_via(Device,Interface,Config),
    type([Device,Cable,Interface],_).
```

Figure C–8: The `configure_device/3` tactic

C.6 Meta-level knowledge

C.6.1 Tactics

The role of the tactics in CLEM is to raise the level of interaction and understanding of the system by the user. For example, the user is usually interested in whether a device *can* be connected into the configuration, not necessarily *how* — if she is interested in how, it is probably at some higher level: *e.g.* whether a fibre-optic link is possible. The details of actually connecting up the device: choosing a cable, *etc.* are usually of interest only to the engineer, although they are obviously important insofar as achieving a fully working system is concerned.

We can think of the tactic as grouping together all the small, detailed steps involved in achieving some higher level goal. For example, see Figure C–8 (pg.219), which shows the tactic for connecting a device. The three predicates used in this tactic have been seen in `types.pl` and `conn.pl`. Let us examine how this tactic works.

Suppose, as is usual, the tactic is called in mode

- `configure_device(+Device,-Interface,-Config)`:

in other words, we want the device to be configured but we don't care how or where.

The first step, carried out by `connect_cable(+Device,-Cable,-Config)`, is to discover whether the device has been allocated a cable, and, if not, to synthesize one.

The second step, carried out by `connected_via(+Device,-Interface,-Config)`, is to connect the device via an interface channel.

The third step is to ensure that this is all legal: in other words, that the device-cable-interface combination is in a type. We do not much care which type, hence the underscore in

```
type([Device,Cable,Interface],_).
```

In fact, it will be something like

```
cross(cross(disk,system_cable),channel)
```

The tactics are to be found in the file `tactics.pl`.

• `configure_processor(?Processor,-Config)`

Usually this is called in mode configure_processor(+Processor,-Config). In theory, we could call it with `Processor` uninstantiated and try different processors in turn. □

• `special(-Config)`

Usually `Config` is a list, the first term of which is ground, and has type `list(processor)`. The rest of the term can then be matched to a template, and `special/1` then configures what is known as the *minimal configuration.* □

• `configure_device(+Device,-Interface,-Config)`

This is used to configure devices as is explained in more detail above; see also Figure C–8 (pg.219). □

• `h_configure_device(+Device,-Interface,-Config)`

This is like `configure_device/3`, except that is uses heuristic limits rather than hard constraints in configuring the device, *i.e.* it will *not* succeed if the device can be configured legally, but only by breaching heuristic limits. □

• `configure_ic(?Interface,-Cardcage,-Config)`

This tactic configures an interface channel (currently a MUX, an HPIB channel, an HPFL (for fibre optic links), or a LANIC, and other various components for connecting serial devices) in a card cage, which can be specified or (more usually) left unspecified. It is analogous to the tactic for configuring devices in interfaces.

Note that `Interface` need not be instantiated. Sometimes we want to say "configure *some* interface" and we don't have any particular one in mind. This reflects user preoccupation with high-level functionality rather than with low-level detail. □

• `h_configure_ic(?Interface,-Cardcage,-Config)`

This is like `configure_ic/3`, except that is uses heuristic limits rather than hard constraints in configuring the interface, *i.e.* it will not succeed if the interface can be configured legally, but only by breaching heuristic limits. □

• `configure_cc(?Cardcage,-Config)`

This configures (some) card cage in a configuration, obeying the constraints operable for that configuration. □

• `match_attributes(+Goal,?Device)`

This can be used:

1. In mode

   ```
   match_attributes(+Goal,-Device)
   ```

   Given a `Goal` which we want a `Device`, as yet unspecified, to achieve (*e.g.* a certain speed of printing, support of given applications), this predicate returns a suitable device.

2. In mode

   ```
   match_attributes(+Goal,+Device)
   ```

   the predicate checks whether the device can achieve the goal.

We needs both modes. For example, given the specification fragment:

```
printers(C,[P]),
ppm(C,P,10),
printer_type(C,P,page)]).
```

a trace of the relevant part of the synthesis of the printer `P` is shown in Figure C–9 (pg.222). `match_attributes/2` is called first (at `1`) in mode (+,–). The printer model `2686A`, which has a suitable speed, is found. Now `match_attributes/2` is called (at `2`) in mode (+,+). It fails, since the `2686A` is not a page printer. On backtracking, the `2680A` is found (at `3`); now the call (at `4`) is successful, since we have a page printer here.

As can be seen from the trace, there are no other printers which meet both requirements. □

• `disk_storage(+Capacity,?Dks,?Config)`

Given a desired capacity in megabytes,

```
disk_storage(+Capacity,+Dks,?Config)
```

checks whether the disk drives of the configuration have at least this capacity, and

```
(1) Call:
      match_attributes(          ppm(_6730,10),        _6730)
    Exit:
      match_attribute            ppm('2686A'(3),10),   '2686A'(3))
(2) Call:
      match_attributes(printer_type('2686A'(3),page),'2686A'(3))
    Fail:
      match_attributes(printer_type('2686A'(3),page),'2686A'(3))
    Redo:
      match_attributes(          ppm('2686A'(3),10),   '2686A'(3))
(3) Exit:
      match_attributes(          ppm('2680A'(4),10),   '2680A'(4))
(4) Call:
      match_attributes(printer_type('2680A'(4),page),'2680A'(4))
    Exit:
      match_attributes(printer_type('2680A'(4),page),'2680A'(4))

X = '2680A'(4) ;

  Redo: match_attributes(printer_type('2680A'(4),page),'2680A'(4))
  Fail: match_attributes(printer_type('2680A'(4),page),'2680A'(4))
  Redo: match_attributes(          ppm('2680A'(4),10),   '2680A'(4))
  Fail: match_attributes(          ppm(_6730,10),        _6730)
no
| ?-
```

Figure C–9: Synthesizing a page printer with a speed of at least 10ppm

```
disk_storage(+Capacity,-Dks,?Config)
```

will generate a disk configuration of such a capacity. The capacity referred to here is that available for user data. □

• total_disk_storage(+Capacity,?Sys,?Dks,?Config)

This predicate is as `disk_storage(+Capacity,?Dks,?Config)`, except that it takes account of the system disk drive(s). It will, if necessary, generate either the user or the system disk drives, or both. □

• tape_capacity(+Capacity,?Tps,?Config)

This predicate is as `disk_storage(+Capacity,?Dks,?Config)`, except for tape (backup) storage rather than disk. The desired capacity must be known, or else something quite random will be synthesized, which, although legal, may be quite useless. Hence the preconditions for the corresponding method (*qv*) are quite strong. □

• configure_device_list(?List,-Config)

This predicate sets a given partial configuration: for example, we may want to specify the printers explicitly as (say) one `2680A` model printer and two `2235A` model printers, in which case the specification fragment will be:

```
printers(Config,['2680A'(_),'2235A'(N1),'2235A'(N2)]),
```

Alternatively, we may want to specify them implicitly; for example, that one has a speed of at least 20ppm, and that another has a speed of at least 450cps and is a line printer. Then the specification fragment will be:

```
printers(C,[P1,P2]),
ppm(C,P1,20),
cps(C,P2,450),
printer_type(C,P2,line)]),
```

`match_attributes/3`

(*qv*) will be used to partially instantiate `P1` and `P2`, and

`configure_device_list/2`

will be sent to work on the goal `printers(C,...)`. It is better if the predicate is called in the mode (+,-); this is achieved through meta-level control where possible. □

```
method(name(...Args...),          % name slot: Prolog term
       G,                         % input slot: (sub)goal
       [...Preconditions...],     % preconditions-slot:
                                  %     % list of conjuncts
       [...Effects...],           % effects-slot:
                                  %     % list of conjuncts
       [...Outputs...],           % output slot:
                                  %     % list of (sub)goals
       tactic(...Args...)         % tactic slot: Prolog term
      ).
```

Figure C–10: The general form of a `method/6` term.

- `is_legal(?Config)`

This predicate checks that all essential components and connections, without which `Config` will not be a legal configuration, are present. □

C.6.2 Methods

Simple Methods

Methods are the building blocks which make up proof plans. They are specifications of tactics, which in turn are Prolog procedures which execute a number of proof steps. A method is a structure with 6 "slots", and is as a Prolog `method/6` term. Each of the slots corresponds to an argument of the `method/6` term, in the order listed above. The general form of a `method/6` term is shown in Figure C–10 (pg.224).

1. The *name-slot* is a Prolog term of the form `name(...args ...)`, corresponding to the name and the arguments of the method. For example the method to configure a device is called `configure_device`. This particular method has three arguments, specifying the device to be configured, the slot in which to configure it, and a third argument for the configuration being gradually synthesized.

2. An *input-slot*, specifying the object-level formula to which the method is applicable. The input-slot is a Prolog term that should unify with the input to which the method applies. Since the mechanism for this is different from that for CLAM, it is explained here.

A specification is a conjunct

$$G_1 \wedge G_2 \wedge \ldots G_n \tag{C.1}$$

If *one* goal G_i unifies with the input slot of a method, then this counts as unification and the preconditions (see below) will be tested on this goal.

We shall see, in Section C.6.2, a generalization of this to supermethods which takes account of more than one input goal.

3. A *preconditions-slot*, specifying conditions that must be true for the method to be applicable. The preconditions-slot is a list of Prolog goals, each of which should succeed after the input-slot has been unified with the input conjunct.

   Note, then, that a method is said to be *applicable* if the input-slot unifies with the input conjunct as explained above, and all of the preconditions are true.

4. The effects-slot is a list of Prolog goals, specifying properties that will hold after the method has applied successfully. If the input has matched, and if the preconditions hold, the effects should always succeed.

5. An *output-slot*, specifying the object-level formulae that will be produced as subgoals when the method has applied successfully. It is a list of subgoals which remain to be proved after the method has been applied to the input. For example, in the case of the `configure_device` method, no goals remain to be solved after the tactic has been applied, so this slot is the empty list. On the other hand, after the method `configure_device_list` has been applied, each one of the devices in the list needs to be separately configured, thus the output goal is `repeat(configure_device(...))`.

6. A *tactic-slot*, giving the name of the tactic for which this method is a specification.

To a certain extent, the effects slot ghosts the running of the tactic, and can be thought of as a "cheap" version of the tactic.

An example of a particular method (the h_configure_device/3 method which will be further discussed in Section C.6.5 (pg.229)) is shown in Figure C–11 (pg.226).

All slots can share Prolog variables. In particular, variables which are bound while unifying the input slot with the input subgoal can be referred to in the preconditions, effects, and output slots.

Methods can be found in the file `methods.pl`.

```
method(h_configure_device(Device,_,C),              % name
    configure(Device,IC,C),                         % input
    [needs_ic(Type2,ICtype),                        % preconditions
     h_slots_available(C,IC,Type,ICtype,N),
     N>0))],
     [reduce_slots(IC,Device,Type,ICtype,1),        % effects
     guess_components(system_device,C,SysDev),
     ((gmember([Device,_],SysDev),!)
      ;add_member([Device,_],SysDev))],
     [],                                            % output
     h_configure_device(Device,_IC,C)).             % tactic
```

Figure C–11: The `configure_device/3` method.

```
method(disk_storage(Cap,Dks,C),          % name
     disk_capacity(C,Cap),               % input
     [guess_components(disk,C,Dks),      % preconds
      \+groundp(Dks)],
     [generate_disks(C,Dks,Cap)],        % effects
     [repeat(configure(Dks,_,C))],       % output
     disk_storage(Cap,Dks,C)).           % tactic
```

Figure C–12: The `disk_storage/3` method.

Supermethods

Supermethods in CLEM are those which call other methods from its effects, and are therefore seen as having other methods as their basic "building blocks". The `basic_config/1` method is an example of such a supermethod.

The top-level predicate for calling a method from the preconditions or effects from another method is `applicable_subm/3`, described in Section C.7 (pg.235). Figure C–13 (pg.227) shows the basic_config/1 supermethod.

Firstly, note that the matching mechanism for the input slot is slightly different for supermethods. Remember that the specification is a conjunct of goals $G_1 \wedge \ldots \wedge G_n$. There must be a goal G_i for *each* goal in a list of goals in the input slot, with the exception that if the goal is itself a list, as in

```
[system_disks(C,Sys),
disk_capacity(C,Cap),
total_disk_capacity(C,TCap)],
```

```
supermethod(basic_config(C),
    Inputconj,
    [processors(C,P),
     [system_disks(C,P),disk_capacity(C,_DCap),total_disk_capacity(C,_TDCa
     printers(C,Prs),
     terminals(C,Ters)],
    [],
    [applicable_subm(...,method(configure_processor(C,P),_,...),

     ...   ...   ...   ...   ...   ...   ...   ...   ...   ...

     iterate(...
             method(match_attributes(C,_,_),_,...)),

     ...   ...   ...   ...   ...   ...   ...   ...   ...   ...


     applicable_subm(...submethod(configure_disks(C),_,...)),

     (applicable_subm(...,submethod(configure_tapes(C),_,...)),

     iterate_set(...,
             [method(h_configure_device(C,_,_),_,_,_,_,_),
              method(h_configure_ic(C,_,_),_,_,_,_,_),
              method(configure_cc(C,_),_,_,_,_,_),
              method(configure_device(C,_,_),_,_,_,_,_)],
                ...),
     applicable_subm(...,method(configure_device_list(C,Prs),
                                                printers(C,Prs),_,.
     applicable_subm(...,method(configure_device_list(C,Ters),
                                              terminals(C,Prs),_,..

     ...   ...   ...   ...   ...   ...   ...   ...   ...   ...

     ],
    Outputconj,
    Tactic).
```

Figure C–13: The basic_config/1 method.

for instance, then we only require that *at least one* such goal be present in the specification (or what remains of it so far). Thus this list structure of the input slot can be read as a conjunct of disjuncts.

If the preconditions hold (empty for this example), then the effects are run. All the effects in this example consist of running through the applicability of various methods and submethod.

Notice also that the planing mechanism for supermethods requires them to have an extra argument, for the remaining input conjunct.

Submethods should be used if the method is not to be used in its own right during the plan formation, but only as a submethod to be called from other methods. Thus the submethod `configure_disks` is visible only from the effects of supermethods which require it, and not by the top-level planning mechanism.

When deciding whether to afford full method status to a submethod, care should be exercised. The key question is whether, if the tight control of its use given by restricting it to the effects of supermethods is removed, any harm will result.

In many cases, however, I have designated certain cases as submethods simply to make it clear that I regard them solely in this light and do not envisage any more independent usage for them.

C.6.3 Representing configuration schemes

The supermethods represented in CLEM correspond to the various configuration schemes I have isolated. `basic_config/1` corresponds to the most straightforward "normal" case, *i.e.* where disks, terminals, and printers are all specified in some way. This encompasses a great number of different cases and is very flexible. Note that the submethods called in the effects slots are themselves supermethods, except that they are not visible to the planner.

Other supermethods correspond to other common cases. No doubt more could be added at a later stage.

C.6.4 The methods database

At the moment the mechanism for choosing which methods and supermethods to load is somewhat crude. The order of the database can be changed by the user, and methods can be added or removed.

The order in which the methods occur in the database is significant. The only planner so far developed, the depth-first planner, as described in Section C.7 (pg.235), tries to apply methods in the order in which they appear in the database.

• `method(?M,?I,?Pre,?Eff,?O,?T)`

This is the main predicate for accessing the elements in the database of methods. These are all visible to the planner. □

- `supermethod(?M,?Inconj?I,?Pre,?Eff,?O,?T)`

As the `method/6` predicate, but for the database of supermethods. These are all visible to the planner and lexically precede all the methods. If none of the configuration schemes represented so far by supermethods is applicable, then a "customized" plan will be put together from the methods, if possible. This allows for "degenerate" cases (*e.g.* specifications without terminals, or disk drives, or printers). □

- `submethod(?M,?I,?Inconj?Pre,?Eff,?O,?T)`

As the `supermethod/7` predicate, but for the database of submethods.However, unlike both methods and supermethods, submethods are all *invisible* to the planner. □

- `list_methods`

Returns a list representing the current order of methods and supermethods in the database. □

C.6.5 Current repertoire of methods and supermethods

- `basic_config(C)`

Supermethod. This deals with the simplest "normal" case. It takes us as far as generating devices and configuring them (somehow). It tends to give configurations which obey heuristic limits for preference, since the appropriate methods occur early in the database, but is not over-fussy. □

- `constraint_relax_I(C)`

Supermethod. Like `basic_config/1` but allows only the heuristic limit for channels to be breached. □

- `constraint_relax_I(C)`

Supermethod. Like `basic_config/1` but allows the heuristic limit for card cages to be breached. □

- `cheap_config(C)`

Supermethod. This ignores heuristics, *i.e.* uses submethods which take account of legality only. □

- `efficient_config(C)`

Supermethod. In contrast to `cheap_config`, this uses submethods which take account of heuristics only. □

- `cost_config(C,Cost)`

Supermethod. This method deals with the cases where there is a goal

$$cost(Config, Cost)$$

in the specification. This is taken to mean: "the cost of the configuration must not be more than $Cost$". C is annotated using an extra argument, initially $Cost - X$. Methods for adding components subtract the cost of that component: the result is not allowed to become negative. Very trivial costs are ignored, which is just as well as usually cables *etc.* do not usually feature at all at the planning stage. □

- `configure_processor(P,C)`

This picks out a processor for the configuration. Its output goal is to find the appropriate "template" configuration. It saves a lot of problems if this is done early. □

- `special(C)`

This configures a template configuration. The preconditions insist that the processor be known. □

- `match_attributes(Config,Goal,Device)`

This method deals with all goals which are about finding devices to meet given criteria, and will match against goals for specifying speeds of printers, applications supported, *etc.* □

- `connect_cable(Device,Cable,Config`

This is used principally (but not exclusively) as a submethod by the `configure_all_cables/1` supermethod. It ensures, in its preconditions, that the device does not already have a cable. □

- `configure_device(Device,Connection,Config)`

The preconditions of this method ensure that the device is not already connected. The effects give a "dummy" slot — in general, `Connection` remains unknown. □

- `h_configure_device(Device,Connection,Config)`

As previous method, but taking into account soft, rather than hard, constraints. □

- `configure_serial_device(Device,Connection,Config)`

This is similar to `configure_device` but for serial device connection. Note that its tactic is the same as `configure_device`. This is an instance of where the tactic name is not the same as the method name. □

- `configure_ic(Interface,Cardcage,Config)`

The counterpart to `configure_device`, but for configuring interfaces in card cages rather than devices in interfaces. However, note that the input slot insists that the current goal be that of configuring a device. This ensures the (implicit) precondition that an interface is only configured if there is likely to be a device configured on it, and empty interfaces do not sprout up everywhere regardless. □

- `h_configure_ic(Interface,Cardcage,Config)`

As previous method, but taking into account soft, rather than hard, constraints. □

- `configure_cc(Cardcage,Config)`

In the context that we are trying to configure a device, this method is used to add another card cage to a configuration. □

- `configure_portgroups(PG,DTC,Config)`

This is similar to `configure_ic` but for serial device connection. It will only be used in the case that a serially connected device is hanging around. Note that its tactic is the same as `configure_ic`. □

- `configure_serial_connection(DTC,Interface,Config)`

This is for configuring the connections necessary to connect the portgroups necessary to connect serial devices ...Serial device connection is *so* complicated! □

- `configure_device_list(List,config)`

This is applicable on any goal of the form

```
device_types(Config,List)
```

and sets up subgoals to configure each of the devices in turn. □

- **`disk_storage(Capacity,Dks,Config)`**

This deals with goals to do with disk capacity. At the moment, this has to be given explicitly. With more domain knowledge, there is no reason why this could not be done implicitly. □

- **`total_disk_storage(Capacity,Sys,Dks,Config)`**

As previous, for all disk drives. □

- **`tape_capacity(Cap,Tps,Config)`**

As above, but for tape storage. However, in this case it is not necessary for the specification to give the tape capacity required. We need to know only the disk capacity. □

- **`is_legal(Config)`**

The preconditions check for the presence of vital components. The effects of this method are empty. Either the preconditions succeed, or another method must be tried, for example one of the next two. □

- **`add_system_disks(Sys,Config)`**

This is only applicable in the event that we are trying to prove the goal `is_legal(Config)`. It imposes storage for the operating system on the configuration. It is very late in the database, to allow a user choice to act, if there is one. □

- **`add_backup(Tps,Config)`**

Much the same applies here. It imposes backup tapes. □

C.6.6 The method language

The method language used within methods is found in `method-lang.pl`, which defines all the predicates used in the preconditions and effects of methods.

- **`groundp(+Term)`**

Succeeds if `Term` is ground, or if its principal functor is. □

- `umember(+El,?List)`

Like `member/2`, except that `El` can only occur once in `List`. Since, in general, `List` has a variable tail, if `El` does not occur in the ground part of the list, it can be "added" to the non-ground part. □

- `gmember(+El,?List)`

Succeeds if `El` is a ground member of the list `List`. □

- `empty(?List)`

Succeeds if `List` is either ground, and the empty list, or if it is wholly variable — "empty so far", as it were. □

- `no_connected(?List,?N)`

`List` is a list which could have a variable tail. `N` counts only the ground elements of `List` — "number connected so far". □

- `forall(+Goal)`

`Goal` is a functor, the first argument of which is a list. `Goal` is called, with the list substituted by each of its elements in turn. □

- `configured(+Comps,?PConfig)`

Succeeds if the components given in `Comps` are configured as part of the partial configuration `PConfig`. Used chiefly to avoid configuring the same things twice! □

- `guess_components(+Type,?Config,?Components)`

Retrieves components configured so far without type checking. □

- `guess_type(?Component,?Type)`

Guesses the type of a component without generating an actual instance. □

- `guess_supertype(?Component,?Type)`

As previous, but for supertypes. □

- `needs_ic(?Type1,?Type2)`

Succeeds if *comp*1 × *comp*2 will be a type, where *comp*1 is of type `Type1` and *comp*2 is of type `Type2` □

- `slots_available(+Comp,+Type,-N)`

Returns the number `N` of slots available on the component `Comp` for the use of components of type `Type`. □

- `h_slots_available(+Comp,+Type,-N)`

As previous, but using heuristic limits to calculate `N`. □

- `reduce_slots(?Comp,+Type,+N)`

Reduces the number of slots available on the component `Comp` for the use of components of type `Type` by `N`. Actually in practice `N` is always 1, but I thought a general predicate might be useful. □

- `make_list(+Component,-List)`

This creates a list of objects all of the form `Component`. This is unbelievably useful for avoiding silly configurations of (say) ten disk drives, all of a different model. □

- `at_ic_per_cc_heur_limit(?Config)`

This checks whether the heuristic limit has been breached. It is rather complicated. According to heuristics, there should not be an *average* of more than three interface cards per card cage. Thus (apparently) if there are two card cages it is quite all right to have five interface cards on one card cage as long as there is only one on the other. I gave it a clumsy name to go with its clumsy use and implementation. □

- `exist_empty(?Comp,?Config)`

This checks whether any components matching with `Comp` are "empty", in other words have nothing connected to them. This is supposed to stop the ridiculous case where we have just configured a new card cage in order not to breach heuristic limits, then go and put our floating interface card in one of the old card cages. After all, (6 + 0) divided by 2 is 3, which is our maximum heuristic "average",

is it not? So we avoid this situation by forcing an empty card cage to be used, giving in the above example $(5 + 1)$ rather than $(6 + 0)$. □

• `run_goal(+Goal,-Instantiated_Goal)`

In some cases, there is no "cheap" effect to substitute for executing the goal, except that we don't want to do expensive type checking and generate instances of components. So we "run" the goal with dummy indexes. □

C.7 The planner

This section will discuss the planner which can be used to construct proof plans for a given theorem using the available methods.

The planner used in CLEM is forward chaining: it starts by looking at the initial goal list, or theorem to be proved:

$$\exists c : \textbf{configuration}\ spec(c)$$

and then tries to find out which methods are applicable (i.e. which methods have a matching input slot and a succeeding preconditions slot). After picking one of these applicable methods the planner computes the output goal list by evaluating the effects slot of the chosen method. This output list will then serve as the input goal for the next recursive cycle of the planner, until a method has been found which terminates the plan (in other words: until a terminating method (a method with an empty output slot) has been found).

In this description of the planning process, a number of choice points occur: often more than one method will be applicable to the input goal and one method may apply in more than one way (i.e. its preconditions may be satisfied in more than one way).

• `plan(+Spec,?Plan)`

`Spec` always takes the form `spec(Config)=Spec`, where `Spec` is a list of goals referencing `Config`. □

As explained above, a crucial step in the planning process is to find out which methods are applicable to a given input goal conjunct. For this purpose, all planners use the same predicate, namely the predicate `applicable/1`.

• `applicable( ?Method)`

Succeeds if `Method` is applicable. This predicate, and the predicates `test conditions` and `apply effects` will not be found to be much different from those used in CLAM (van Harmelen, 1989) and will not be detailed here. □

```
specification
     disk_capacity(_6735,500)
     processors(_6735,[950(_6744)])

THIS IS THE PLAN
---------------
configure_processor([950(_6744)],S)
special(S)
disk_storage(500,[7937H(_7507)],S)
h_configure_device(7937H(_7507),_7575,S)
add_system_disks([7937H(_8227)],S)
h_configure_device(7937H(_8227),_8341,S)
add_backup([7979A(_8997)],S)
h_configure_device(7979A(_8997),_9081,S)
is_legal(S)
```

Figure C–14: An simple example of a CLEM plan.

• `applicable_subm(+Inputconj, ?Method, Outputconj)`

Succeeds if `Method` is applicable. Like `applicable/1` but with extra machinery for making sure that the modified input conjunct is propagated through to the next stage. □

C.8 Utilities

C.8.1 Applying plans

The products of CLEM's planners are only plans for proofs, they are not proofs themselves. In order to produce a proof, we have to apply a plan in the object-level logic.

After a plan has been constructed by one of the planners, it can be executed to synthesize a configuration meeting the given specification by executing the tactics corresponding to each of the methods.

As described in Section C.6.1 (pg.219), plans can be executed simply by passing them as an argument to CLEM's `execute/1` predicate, subject to slight amendment to take account of the fact that some methods employ the same tactic, and to allow the configuration argument to be passed correctly.

• `apply_plan(+Plan)`

This predicate applies `Plan`, and makes each method in `Plan` a single step in the proof. Progress of the plan execution process can be monitored using the tracing package (tracing level 1 or above). □

• `nat(+Model,?N)`

The predicate `nat/2` can be called in two modes, `nat(+Model,+N)`, when it simply checks that `N` is a natural number and succeeds, or `nat(+Model,-N)`. In the latter case, an index `N` is generated, unique to the model `Model`: *i.e.* a new component is created. The mechanism for doing this is explained in Section C.8 (pg.236). □

This is used, when executing plans, to generate a unique component of the required kind.

C.8.2 Tracing planners

CLEM provides a very simple tracing package that allows the user to monitor the activities of the planners during the planning process. The user can set a tracing level, using the predicate:

• `trace_plan(?Current,?New)`

Currently implemented tracing levels are:

0 No tracing.

1 Prints which methods are being tested for applicability. Also prints out which methods are being applied, on execution.

3 Prints when preconditions and effects of methods succeed.

This mechanism again is similar to the one in CLAM (van Harmelen, 1989).

□

C.8.3 Statistics package

The simplest way of collecting statistics on the behaviour of CLEM is the predicate `runtime/[2;3]`:

- `runtime(+Pred, ?Time)`

This will execute `Pred` as a Prolog predicate, and if successful, will unify `Time` with the CPU time spent while executing `Pred`, measured in milliseconds. This measurement is notoriously unreliable on Unix systems (especially when `Time` is small). Therefore, it is often better to use the predicate `runtime/3`, which runs goals several times and takes the average. □

- `runtime(+Pred, +N, ?Time)`

This will execute `Pred` `N` times, and if successful, will unify `Time` with the average CPU time spent while executing a call to `Pred`. The larger `N` and `Time` are, the more reliable the value of `Time` will be. □

C.8.4 Debugging utilities

A simple tracing package has been implemented to help debugging and using CLEM. This package is described in Section C.8.2 (pg.237). The predicate that should be used to introduce more trace points in newly constructed code are the predicate `moreguff/0`. Less is obtained using `lessguff/0`. We can also use `guff(off)` and `wallpaper/0` for the maximum level. As you might guess, these facilities are not very sophisticated as yet.

In addition, there are predicates such as `list_components/0` which give lists of all components currently stored (using the predicate `type/2`), and `list_methods/0`, which gives all methods currently loaded, in the order in which they will be searched.

C.9 Getting started with CLEM

CLEM runs in Quintus Prolog. To get started, type:

```
[clem].
```

Specifications have the form

```
processors(C,['950'(_)]),
disk_capacity(C,500),
lpm(C,P1,500),
printers(C,[P1]),
system_disks(C,['7937FL'(_)]),
tapes(C,['7980XC'(_)]).
```

There are a number of tests in the file `tests.pl`. To run these, type either

```
config(Specname).
```

with Specname instantiated to one of the tests, *e.g.* `spec1`, or else

```
batch(Specname).
```

The first of these allows you to interact with CLEM, so that you may choose which plan to execute, and whether to backtrack for further solutions on backtracking. Figure C–15 (pg.240) gives an example. *batch/1* will simply find one plan, and execute it. There is no interaction with the user.

You may also choose to run

```
time(Specname).
```

for statistics gathering. However, it is better to use the special `stats` subdirectory, with versions of the files which suppress output.

In addition, various predicates exist for benchmarking purposes. See the documentation in `tests.pl` for details.

```
| ?- config(spec1).

THIS IS THE PLAN
----------------
configure_processor(S,[950(_6913)])
special(S)
add_system_disks(S,[7937H(_7746)])
h_configure_device(S,7937H(_7746),_7855)
add_backup(S,[7980A(_8506)])
h_configure_device(S,7980A(_8506),_8587)
is_legal(S)


Execute this plan(y/n)?n.

THIS IS THE PLAN
----------------
configure_processor(S,[950(_6913)])
special(S)
add_system_disks(S,[7937H(_7746)])
h_configure_device(S,7937H(_7746),_7855)
add_backup(S,[7980A(_8506)])
configure_device(S,7980A(_8506),_8587)
is_legal(S)


Execute this plan(y/n)?y.

EXECUTING
---------

            < gives configuration >

Happy(y/n)?y.

yes
| ?-
```

Figure C–15: An example of CLEM-user dialogue.

C.10 The organization of the source files

Object-level code is in

- `types.pl`
- `config-attr.pl`
- `axioms.pl`
- `conn.pl`
- `constraints.pl`

Heuristic knowledge is found in

- `heuristics.pl`

Meta-level knowledge and meta-level support are found in

- `tactics.pl`
- `methods.pl`
- `method-lang.pl`
- `planner.pl`

General utilities are found in

- `utils.pl`

and planning utilities (tracing, statistics, *etc.*) in

- `planning-utils.pl`

Appendix D

Selected Code

D.1 Object-level theory

D.1.1 Types

```
 1:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 2:          %                                    %
 3:          %          types.pl                  %
 4:          %        last updated                %
 5:          %        7th May, 1991               %
 6:          %                                    %
 7:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 8:
 9: %%% This file contains
10: %%%       The types of all components in the
11: %%%        intitial subset used.
12: %%%       Rules for forming compound objects.
13:
14: %%% Mostly I use the product numbers: thus '7937H' refers
15: %%% to a model of disk. However, there are cases where I
16: %%% do not know these, usually because the component is
17: %%% most often found as part of a 'bundle'. In these cases,
18: %%% I have used a suitable mnemonic, e.g. 'mux', 'mem', etc.
19:
20: %%% These clauses are all of the form
21: %%% type(comp-name(index),comp-type):-nat(comp-name,index).
22: %%% For example: type('950'(N),processor)
23: %%%  holds provided N is a natural number.
24:
25: %%% note the function of the predicate
26: %%%          nat(+component,?index)
27: %%%  if index is not instantiated, then a unique
28: %%%  index is created for this component type.
29: %%% Thus is can be used to check, or generate, a uniquely
30: %%% referenced component.
31:
```

```
32:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
33:           %                                   %
34:           %     processor                     %
35:           %                                   %
36:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
37:
38: type('950'(N),processor):- nat('950',N).
39:
40:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
41:           %                                   %
42:           %     memory modules                %
43:           %  (only one size for the 950)      %
44:           %                                   %
45:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
46:
47: type(mem(N),memory):- nat(mem,N).
48:   %%% component number unknown just now
49:
50:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
51:           %                                   %
52:           %     cardcage                      %
53:           %                                   %
54:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55:
56: type('950-cardcage'(N),cardcage):- nat('950-cardcage',N).
57:   %%% component number unknown just now
58:
59:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
60:           %                                   %
61:           %     channel                       %
62:           %                                   %
63:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
64:
65: type(hpib(N),channel):- nat(hpib,N).
66:   %%% component number unknown just now
67:
68:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
69:           %                                   %
70:           %     fchannel                      %
71:           %                                   %
72:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
73:
74: type(hpfl(N),fchannel):- nat(hpfl,N).
75:   %%% component number unknown just now
76:
77:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
78:           %                                   %
79:           %     lanic                         %
80:           %                                   %
81:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
82:
83: type(lnc(N),lanic):- nat(lnc,N).
```

```
 84:   %%% component number unknown just now
 85:
 86:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 87:           %                                  %
 88:           %     mux                          %
 89:           %                                  %
 90:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 91:
 92: type(mx(N),mux):- nat(mx,N).
 93:   %%% component number unknown just now
 94:
 95:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 96:           %                                  %
 97:           %     disk drives                  %
 98:           %                                  %
 99:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
100:
101: type('7937H'(N),disk):- nat('7937H',N).
102: type('7937XP'(N),disk):- nat('7937XP',N).
103: type('7933H'(N),disk):- nat('7933H',N).
104: type('7933XP'(N),disk):- nat('7933XP',N).
105: type('7935H'(N),disk):- nat('7935H',N).
106: type('7935XP'(N),disk):- nat('7935XP',N).
107: type('7936H'(N),disk):- nat('7936H',N).
108: type('7936XP'(N),disk):- nat('7936XP',N).
109:
110:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
111:           %                                  %
112:           % fdisk (fibre optic disk drives) %
113:           %                                  %
114:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
115:
116: type('7937FL'(N),fdisk):- nat('7937FL',N).
117: type('7936FL'(N),fdisk):- nat('7936FL',N).
118:
119:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
120:           %                                  %
121:           %     rs232_portgroup              %
122:           %                                  %
123:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
124:
125: type(rs232ports(N),rs232_portgroup):- nat(rs232ports,N).
126:   %%% component number unknown just now
127:
128:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
129:           %                                  %
130:           %     rs422_portgroup              %
131:           %                                  %
132:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
133:
134: type(rs422ports(N),rs422_portgroup):- nat(rs422ports,N).
135:   %%% component number unknown just now
```

```
136:
137:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
138:          %                                  %
139:          %     modem_portgroup              %
140:          %                                  %
141:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
142:
143: type(modemports(N),modem_portgroup):- nat(modemports,N).
144:   %%% component number unknown just now
145:
146:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
147:          %                                  %
148:          %     tape                         %
149:          %                                  %
150:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
151:
152: type('7979A'(N),tape):- nat('7979A',N).
153: type('7980A'(N),tape):- nat('7980A',N).
154: type('7980XC'(N),tape):- nat('7980XC',N).
155:
156:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
157:          %                                  %
158:          %     printer                      %
159:          %                                  %
160:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
161:
162: type('2563A'(N),printer):- nat('2563A',N).
163: type('2563B'(N),printer):- nat('2563B',N).
164: type('2564A'(N),printer):- nat('2564A',N).
165: type('2564B'(N),printer):- nat('2564B',N).
166: type('2566A'(N),printer):- nat('2566A',N).
167: type('2566B'(N),printer):- nat('2566B',N).
168: type('2567B'(N),printer):- nat('2567B',N).
169: type('2680A'(N),printer):- nat('2680A',N).
170: type('2686A'(N),printer):- nat('2686A',N).
171: type('2688A'(N),printer):- nat('2688A',N).
172: type('2235A'(N),printer):- nat('2235A',N).
173: type('2584A'(N),printer):- nat('2584A',N).
174: type('2586A'(N),printer):- nat('2586A',N).
175: type('2932A'(N),printer):- nat('2932A',N).
176: type('2934A'(N),printer):- nat('2934A',N).
177: type('2235A'(N),printer):- nat('2934A',N).
178:
179:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
180:          %                                  %
181:          %     serial_conn                  %
182:          %                                  %
183:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
184:
185: type(dtc(N),serial_conn):- nat(dtc,N).
186:   %%% component number unknown just now
187:
```

```
188:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
189:          %                                    %
190:          %     terminal                       %
191:          %                                    %
192:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
193:
194: type('2392A'(N),terminal):- nat('2392A',N).
195: type('2394A'(N),terminal):- nat('2394A',N).
196: type('D1347A'(N),terminal):- nat('D1347A',N).
197: type('D1127A'(N),terminal):- nat('D1127A',N).
198:
199:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
200:          %                                    %
201:          %     system_cable                   %
202:          %                                    %
203:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
204:
205: type('hpib-cable'(N),system_cable):- nat('hpib-cable',N).
206:   %%% component number unknown just now
207:
208:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
209:          %                                    %
210:          %     flink                          %
211:          %                                    %
212:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
213:
214: type('hpfl-cable'(N),flink):- nat('hpfl-cable',N).
215:   %%% component number unknown just now
216:
217:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
218:          %                                    %
219:          %    rs232_serial_cable              %
220:          %                                    %
221:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
222:
223: type('40242X'(N),rs232_serial_cable):- nat('40242X',N).
224: type('40242G'(N),rs232_serial_cable):- nat('40242G',N).
225: type('13242X'(N),rs232_serial_cable):- nat('13242X',N).
226: type('24542G'(N),rs232_serial_cable):- nat('24542G',N).
227: type('24542M'(N),rs232_serial_cable):- nat('24542M',N).
228: type(mod(N),rs232_serial_cable):- nat(mod,N).
229:   %%% component number unknown just now
230:
231:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
232:          %                                    %
233:          %    rs422_serial_cable              %
234:          %                                    %
235:          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
236:
237: type('40242P'(N),rs422_serial_cable):- nat('40242P',N).
238: type('13242P'(N),rs422_serial_cable):- nat('13242P',N).
239:
```

```
240:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
241:            %                                    %
242:            %        list types                  %
243:            %                                    %
244:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
245:
246: %%% We can guess that a variable is a list.
247: type(L,list(_)):-
248:     var(L),
249:     !.
250: %%% The empty list is a list.
251: type([],list(_)).
252: %%% A list of objects all the same type Type
253: %%%  is a list(Type).
254: type([H|L], list(Type)):-
255:         type(H,Type),
256:         type(L,list(Type)).
257:
258: %%% If we connect objects of certain types together
259: %%%  we get well-formed objects of type Type1xType2
260: %%% These are the product types in this domain.
261:
262:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
263:            %                                    %
264:            %        product types               %
265:            %                                    %
266:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
267:
268: %%% Some printers will not fall into the category below
269: %%% with Cable = mod(_),
270: %%% as they are unsuitable for modem connection.
271: type([Dev,mod(_)],cross(device,serial_cable)):-
272:     type(Dev,printer),
273:     \+modem_suitable(Dev),
274:     !,
275:     fail.
276:
277: %%% Printers and terminals can be connected via serial cables.
278: type([Dev,Cable],cross(device,serial_cable)):-
279:     (type(Dev,terminal);type(Dev,printer)),
280:     supertype(Cable,serial_cable).
281:
282: %%% Disks, tapes, and printers can be connected via system cables.
283: type([Dev,Cable],cross(device,system_cable)):-
284:     (type(Dev,disk);type(Dev,tape);type(Dev,printer)),
285:     type(Cable,system_cable).
286:
287: %%% Devices connected via fibre-optic links.
288: type([Dev,Cable],cross(fdisk,flink)):-
289:     type(Dev,fdisk),
290:     type(Cable,flink).
291:
```

```
292: %%% Device-cable pairs connected via serial portgroups.
293: type([Dev,Cable,PG],cross(cross(device,cable),conn)):-
294:     type(Dev,Type),
295:     (Type = printer; Type = terminal),
296:     type(PG,rs232_portgroup),
297:     type(Cable,rs232_serial_cable).
298: type([Dev,Cable,PG],cross(cross(device,cable),conn)):-
299:     type(Dev,Type),
300:     (Type = printer; Type = terminal),
301:     type(PG,rs422_portgroup),
302:     type(Cable,rs422_serial_cable).
303:
304: %%% Device-cable pairs connected via interface channels.
305: type([Dev,Cable,IC],cross(cross(device,cable),conn)):-
306:     type(Dev,Type),
307:     (Type = printer; Type = disk; Type = tape),
308:     type(IC,channel),
309:     type(Cable,system_cable).
310:
311: %%% Device-fibre-optic link  pairs connected via fibre-optic channels.
312: type([Dev,Cable,IC],cross(cross(device,cable),conn)):-
313:     type(Dev,fdisk),
314:     type(IC,fchannel),
315:     type(Cable,flink).
316:
317:
318: %%% Lists of interfaces connected via cardcages.
319: type([CC,Lm,Ll,Lc,Lf],cross([CC,Lm,Ll,Lc,Lf])):-
320:     type(CC,cardcage),
321:     type(Lm,list(mux)),
322:     type(Ll,list(lanic)),
323:     type(Lc,list(channel)),
324:     type(Lf,list(fchannel)).
325:
326: %%% Lists of terminals connected via muxes.
327: type([M,Ter],mux_term):-
328:     type(M,mux),
329:     ((var(Ter),!);type(Ter,list(terminal))).
330:
331: %%% Lists of dtcs connected via lanics.
332: type([L,Dtcs],lanic_term):-
333:     type(L,lanic),
334:     ((var(Dtcs),!);type(Dtcs,list(dtc_term))).
335:
336: type([Dtc,Ports],dtc_term):-
337:     type(Dtc,serial_conn),
338:     ((var(Ports),!);supertype(Ports,list(portgroup))).
339:
340: %%% Lists of device-cable pairs connected via portgroups.
341: %%% (add device things here)
342: type([Pg,Devs],portgroup_term):-
343:     supertype(Pg,portgroup),
```

```
344:     ((var(Devs),!);supertype(Devs,list(serial_device))).
345:
346: %%% Lists of devices connected via interface channels.
347: type([Ch,D,T,P],channel_term):-
348:     type(Ch,channel),
349:     ((var(D),!);type(D,list(disk))),
350:     ((var(T),!);type(T,list(tape))),
351:     ((var(P),!);type(P,list(printer))).
352:
353: %%% Lists of devices connected via interface fibre-optic links.
354: type([Fch,D],fchannel_term):-
355:     type(Fch,fchannel),
356:     ((var(D),!);type(D,list(fdisk))).
357:
358: %%% A few supertype defintions.
359:
360:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
361:           %                                    %
362:           %     disk supertype                 %
363:           %                                    %
364:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
365:
366: supertype(Dev,sdisk):-
367:     type(Dev,disk)
368:     ;
369:     type(Dev,fdisk).
370:
371:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
372:           %                                    %
373:           %     serial device supertype        %
374:           %                                    %
375:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
376:
377: supertype(Dev,serial_device):-
378:     type(Dev,terminal)
379:     ;
380:     type(Dev,printer).
381:
382:
383:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
384:           %                                    %
385:           %       device supertype             %
386:           %                                    %
387:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
388:
389: supertype(Dev,device):-
390:     supertype(Dev,sdisk)
391:     ;
392:     type(Dev,tape)
393:     ;
394:     supertype(Dev,serial_device).
395:
```

```
396:
397:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
398:            %                                    %
399:            %    portgroups supertype            %
400:            %                                    %
401:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
402:
403: supertype(G,portgroup):-
404:     type(G,rs232_portgroup)
405:     ;
406:     type(G,rs422_portgroup)
407:     ;
408:     type(G,modem_portgroup).
409:
410:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
411:            %                                    %
412:            %    serial_cable supertype          %
413:            %                                    %
414:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
415:
416: supertype(C,serial_cable):-
417:     type(C,rs232_serial_cable)
418:     ;
419:     type(C,rs422_serial_cable).
420:
421:
422:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
423:            %                                    %
424:            %       cable supertype              %
425:            %                                    %
426:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
427:
428: supertype(C,cable):-
429:     supertype(C,serial_cable)
430:     ;
431:     type(C,flink)
432:     ;
433:     type(C,system_cable).
434:
435:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
436:            %                                    %
437:            %     interface supertype            %
438:            %                                    %
439:            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
440:
441: supertype(IC,interface):-
442:     type(IC,mux)
443:     ;
444:     type(IC,lanic)
445:     ;
446:     type(IC,channel)
447:     ;
```

```
448:     type(IC,fchannel).
449:
450:
451: %%% Allows lists of objects of different types,
452: %%%  provided they are all of the same supertype.
453:
454:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
455:           %                                   %
456:           %     list supertype                %
457:           %                                   %
458:           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
459:
460: supertype(L,list(_)):-
461:     var(L),
462:     !.
463: supertype([],list(_)).
464: supertype([H|L], list(Type)):-
465:         supertype(H,Type),
466:         supertype(L,list(Type)).
467:
468:
```

D.2 Meta-level knowledge

D.2.1 Tactics

```
  1:              %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  2:              %                                   %
  3:              %         Tactics.pl                %
  4:              %        Last Updated               %
  5:              %         8th March                 %
  6:              %                                   %
  7:              %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  8:
  9: %%% configure_processor(-C,+P)
 10: %%% So far we have used this tactic in this mode only, althoug
 11: %%% configure_processor(-C,-P) is feasible.
 12: configure_processor(C,P):-
 13:     processors(C,P).
 14:
 15: %%% special(-C)
 16: %%% C is a list: the first item must be ground, or
 17: %%% instantiable to a processor this system knows about.
 18: special(C):-
 19: C= [['950'(N)],_Sys,_Ds,_Ts,_Prs,_Cons,_Ters,
 20:                 _CC,_Ms,_Ls,_Pgs,_Cs,_Fs,
 21:                  _SysDev,_FlDev,_SerDev,_CCtl],
 22:     processors(C,['950'(N)]),
 23:     configure_cc(C,'950-cardcage'(CC1)),
 24:     configure_cc(C,'950-cardcage'(CC2)),
```

```
25:     configure_ic(C,mx(_Mx1),'950-cardcage'(CC1)),
26:     configure_ic(C,mx(_Mx2),'950-cardcage'(CC1)),
27:     configure_ic(C,lnc(_Lnc),'950-cardcage'(CC1)),
28:     configure_ic(C,hpib(_Ch1),'950-cardcage'(CC1)),
29:     configure_ic(C,hpib(_Ch2),'950-cardcage'(CC2)).
30:
31: %%% configure_device(-C,+Device,?IC)
32: %%% configures a device on some interface channel
33: %%% according to legal (hard) limits.
34:     % find or generate cable
35:     % if ic ground try to put it there; or find an ic
36:     % (made consistent with other subterms)
37: configure_device(C,Device,IC):-
38:     connect_cable(Device,Cable,C),
39:     connected_via(Device,IC,C),
40:     type([Device,Cable,IC],_).
41: %%% h_configure_device(-C,+Device,?IC)
42: %%% configures a device on some interface channel
43: %%% according to heuristic limits.
44: h_configure_device(C,Device,IC):-
45:     connect_cable(Device,Cable,C),
46:     h_connected_via(Device,IC,C),
47:     type([Device,Cable,IC],_).
48:
49: %%% configure_serial_device(-C,+Device,?IC)
50: %%% Configures a serial device.
51: configure_serial_device(C,Device,IC):-
52:     configure_device(C,Device,IC).
53:
54: %%% configure_ic(-C,+Ch,?CC)
55: %%% Configures a channel interface in a card cage
56: %%% according to legal limits
57:     % if CC ground try and put it there; or find a cc
58: configure_ic(C,Ch,CC):-
59:     add_component(C,Ch),
60:     connected_via(Ch,CC,C).
61:
62: %%% h_configure_ic(-C,+Ch,?CC)
63: %%% Configures a channel interface in a card cage
64: %%% according to heuristic limits
65:     % if CC ground try and put it there; or find a cc
66: h_configure_ic(C,Ch,CC):-
67:     add_component(C,Ch),
68:     h_connected_via(Ch,CC,C).
69:
70: %%% configure_cc(-C,+CC)
71: %%% Add a card cage to a configuration
72: configure_cc(C,CC):-
73:     add_component(C,CC),
74:     cc_terms(C,CCt),
75:     umember([CC,_,_,_,_],CCt).
76:
```

```
 77: %%% configure_serial_connection(-C,+Dtc,?Lnc)
 78: %%% This will configure a DTC on a LANIC.
 79: configure_serial_connection(C,Dtc,Lnc):-
 80:     connected_via(Dtc,Lnc,C).
 81:
 82: %%% configure_portgroups(-C,+G,?Dtc)
 83: %%% This will configure a group of ports on a DTC.
 84: %%% The DTC is connected (somewhere) in the configuration.
 85: configure_portgroups(C,G,Dtc):-
 86:     connected_via(Dtc,_,C),
 87:     connected_via(G,Dtc,C).
 88:
 89: %%% match_attributes(+G,?Dev)
 90: %%% Given a goal G, a Device is found which achieves it.
 91: %%% Should this be here?
 92: match_attributes(G,Dev):-
 93:     find_match(G,Dev),
 94:     type(Dev,_).
 95:
 96: %%% disk_storage(-C,+Cap,?Dks)
 97: %%% Given a capacity for disk storage, checks or generates
 98: %%% disks to meet this.
 99: disk_storage(C,Cap,Dks):-
100:     disks(C,Dks),              % find/generate disks of C
101:     capacity(Dks,Capd),        % check capacity
102:     Capd >= Cap.
103:
104: %%% total_disk_storage(-C,+Cap,?Sys,?Dks)
105: %%% Given a total capacity for disk storage
     %%% plus that needed for the O/S,
106: %%% check or generate appropriate user and system disks.
107: total_disk_storage(C,Cap,Sys,Dks):-
108:     system_disks(C,Sys),
109:     disks(C,Dks),
110:     capacity(Sys,C1),
111:     capacity(Dks,C2),
112:     Cap =< C1 + C2.
113:
114: %%% tape_capacity(C,Cap,Tps)
115: %%% Given a capacity for tape storage, checks or generates
116: %%% tapes to meet this.
117: tape_capacity(C,Cap,Tps):-
118:     tapes(C,Tps),              % find/generate tapes of C
119:     capacity(Tps,Capt),        % check capacity
120:     Capt >= Cap.
121:
122: %%% configure_device_list(-C,?L)
123: %%% Configures a list of devices in C.
124: %%% - put corresponding term to L (or check it)
125: configure_device_list(C,L):-
126:     ((type(L,list(disk)),disks(C,L))
127:      ;
```

```
128:      (type(L,list(fdisk)),disks(C,L))
129:      ;
130:      (type(L,list(printer)),printers(C,L))
131:      ;
132:      (type(L,list(terminal)),terminals(C,L))
133:      ;
134:      (type(L,list(tape)),tapes(C,L))).
135:
136: %%% is_legal(?C)
137: %%% This checks that all essential devices are present.
138: %%% This part does not need to check for connections.
139: is_legal(C):-
140:     processors(C,P),
141:     groundp(P),
142:     system_disks(C,Sys),
143:     groundp(Sys),
144:     tapes(C,Tps),
145:     groundp(Tps).    % add console and memory at some stage
146:
147: %%% add_system_disks(-C,-Sys)
148: %%% Generate system disks for C
149: add_system_disks(C,Sys):-
150:     system_disks(C,Sys).
151:
152: %%% add_backup(-C,-Tps)
153: %%% Generate tapes for C
154: add_backup(C,Tps):-
155:     tapes(C,Tps).
```

D.2.2 Methods

D.2.3 Method Language

```
  1:                    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  2:                    %                                    %
  3:                    %          Methods.pl                %
  4:                    %      Updated 14th March            %
  5:                    %                                    %
  6:                    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  7:
  8:
  9: supermethod(basic_config(C),
 10:      Inputconj,
 11:     [processors(C,P),
 12:     [system_disks(C,P),disk_capacity(C,_DCap),
 13:                                  total_disk_capacity(C,_TDCap)],
 14:      printers(C,Prs),terminals(C,Ters)],
 15:     [],
 16:     [applicable_subm(Inputconj,
 17:                      method(configure_processor(C,P),
 18:                             _,_,_,_,Tac1),
```

```
19:                       Outconj1),
20:      applicable_subm(Outconj1,
21:                       method(special(C),
22:                              _,_,_,_,Tac2),
23:                       Outconj2),
24:      iterate(Outconj2,
25:              method(match_attributes(C,_,_),_,_,_,_,_),
26:              Outconj3,
27:              Tac3),
28:      iterate(Outconj3,
29:              method(connect_cable(C,_,_),_,_,_,_,_),
30:              Outconj4,
31:              Tac4),
32:      applicable_subm(Outconj4,
33:                       submethod(configure_disks(C),
34:                                 Outconj4,_,_,_,_,Tac5),
35:                       Outconj5),
36:      (applicable_subm(Outconj5,
37:                       submethod(configure_tapes(C),
38:                                 Outconj5,_,_,_,_,Tac6),
39:                       Outconj6)
40:      ;
41:      applicable_subm(Outconj5,
42:                       method(add_backup(C,_),
43:                                 _,_,_,_,Tac6),
44:                       Outconj6)
45:      ),
46:      iterate_set(Outconj6,
47:              [method(h_configure_device(C,_,_),_,_,_,_,_),
48:               method(h_configure_ic(C,_,_),_,_,_,_,_),
49:               method(configure_cc(C,_),_,_,_,_,_)],
50:              Outconj7,
51:              Tac7),
52:      applicable_subm(Outconj7,
53:                       method(configure_device_list(C,Prs),
54:                                 printers(C,Prs),_,_,_,Tac8),
55:                       Outconj8),
56:      applicable_subm(Outconj8,
57:                       method(configure_device_list(C,Ters),
58:                                 terminals(C,Ters),_,_,_,Tac9),
59:                       Outputconj),
60:      flatten([Tac1,Tac2,Tac3,Tac4,Tac5,Tac6,Tac7,Tac8,Tac9],Tactic)
61:      ],
62:     Outputconj,
63:     Tactic).
64: supermethod(constraint_relax_I(C),
65:      Inputconj,
66:     [processors(C,P),
67:      [system_disks(C,P),disk_capacity(C,_DCap),
68:                                    total_disk_capacity(C,_TDCap)],
69:      printers(C,Prs),terminals(C,Ters)],
70:     [],
```

```
71:     [applicable_subm(Inputconj,
72:                      method(configure_processor(C,P),
73:                             _,_,_,_,Tac1),
74:                      Outconj1),
75:      applicable_subm(Outconj1,
76:                      method(special(C),
77:                             _,_,_,_,Tac2),
78:                      Outconj2),
79:      iterate(Outconj2,
80:              method(match_attributes(C,_,_),_,_,_,_,_,_),
81:              Outconj3,
82:              Tac3),
83:      iterate(Outconj3,
84:              method(connect_cable(C,_,_),_,_,_,_,_,_),
85:              Outconj4,
86:              Tac4),
87:      applicable_subm(Outconj4,
88:                      submethod(configure_disks(C),
89:                                Outconj4,_,_,_,_,_,Tac5),
90:                      Outconj5),
91:      (applicable_subm(Outconj5,
92:                      submethod(configure_tapes(C),
93:                                Outconj5,_,_,_,_,_,Tac6),
94:                      Outconj6)
95:      ;
96:      applicable_subm(Outconj5,
97:                       method(add_backup(C,_),
98:                              _,_,_,_,_,Tac6),
99:                      Outconj6)
100:     ),
101:     iterate_set(Outconj6,
102:             [method(h_configure_device(C,_,_),_,_,_,_,_,_),
103:              method(h_configure_ic(C,_,_),_,_,_,_,_,_),
104:              method(configure_cc(C,_),_,_,_,_,_,_),
105:              method(configure_device(C,_,_),_,_,_,_,_,_),],
106:             Outconj7,
107:             Tac7),
108:     applicable_subm(Outconj7,
109:                     method(configure_device_list(C,Prs),
110:                                printers(C,Prs),_,_,_,_,Tac8),
111:                     Outconj8),
112:     applicable_subm(Outconj8,
113:                     method(configure_device_list(C,Ters),
114:                                terminals(C,Ters),_,_,_,_,Tac9),
115:                     Outputconj),
116:     flatten([Tac1,Tac2,Tac3,Tac4,Tac5,Tac6,Tac7,Tac8,Tac9],Tactic)
117:     ],
118:    Outputconj,
119:    Tactic).
120: supermethod(constraint_relax_II(C),
121:      Inputconj,
122:     [processors(C,P),
```

```
123:      [system_disks(C,P),disk_capacity(C,_DCap),
124:                                          total_disk_capacity(C,_TDCap)],
125:      printers(C,Prs),terminals(C,Ters)],
126:     [],
127:     [applicable_subm(Inputconj,
128:                      method(configure_processor(C,P),
129:                             _,_,_,_,Tac1),
130:                      Outconj1),
131:      applicable_subm(Outconj1,
132:                      method(special(C),
133:                             _,_,_,_,Tac2),
134:                      Outconj2),
135:      iterate(Outconj2,
136:              method(match_attributes(C,_,_),_,_,_,_,_,_),
137:              Outconj3,
138:              Tac3),
139:      iterate(Outconj3,
140:              method(connect_cable(C,_,_),_,_,_,_,_,_),
141:              Outconj4,
142:              Tac4),
143:      applicable_subm(Outconj4,
144:                      submethod(configure_disks(C),
145:                                Outconj4,_,_,_,_,_,Tac5),
146:                      Outconj5),
147:      (applicable_subm(Outconj5,
148:                      submethod(configure_tapes(C),
149:                                Outconj5,_,_,_,_,_,Tac6),
150:                      Outconj6)
151:      ;
152:      applicable_subm(Outconj5,
153:                       method(add_backup(C,_),
154:                              _,_,_,_,_,Tac6),
155:                      Outconj6)
156:      ),
157:      iterate_set(Outconj6,
158:              [method(h_configure_device(C,_,_),_,_,_,_,_,_),
159:               method(h_configure_ic(C,_,_),_,_,_,_,_,_),
160:               method(configure_cc(C,_),_,_,_,_,_,_),
161:               method(configure_ic(C,_,_),_,_,_,_,_,_),],
162:              Outconj7,
163:              Tac7),
164:      applicable_subm(Outconj7,
165:                      method(configure_device_list(C,Prs),
166:                                printers(C,Prs),_,_,_,_,Tac8),
167:                      Outconj8),
168:      applicable_subm(Outconj8,
169:                      method(configure_device_list(C,Ters),
170:                                terminals(C,Ters),_,_,_,_,Tac9),
171:                      Outputconj),
172:      flatten([Tac1,Tac2,Tac3,Tac4,Tac5,Tac6,Tac7,Tac8,Tac9],Tactic)
173:      ],
174:     Outputconj,
```

```
175:     Tactic).
176:
177: method(configure_processor(C,P),
178:     processors(C,P),
179:     [groundp(P),                              % processor given by spec
180:      \+((C=[P1|_],groundp(P1),\+P=P1))],% consistent with config
181:     [C=[P|_]],                                % instantiate processor
182:     [template(P,C)],                          % and do its template next
183:     configure_processor(C,P)).
184:
185: method(special(C),
186:     template(P,C),
187:     [C=[P|_],groundp(P)],          % must know what the processor is
188:     [find_template(C)],            % form skeleton config
189:     [],
190:     special(C)).
191:
192: method(match_attributes(C,G,Dev),
193:      G,
194:      [G=..[Att,C,Dev,_Value],
195:      (Att = lpm; Att = cps; Att = throughput;
196:       Att=ppm; Att=printer_type)
197:      ],
198:      [find_match(G,Dev)],
199:      [],
200:      match_attributes(G,Dev)).
201:
202: method(connect_cable(C,Device,Cable),
203:      cabletype(Device,Cabletype,C),
204:      [needs_ic(Cabletype,Ictype),   % find out connection type
205:       \+((guess_portgroups(C,Pgs),  % each device has exactly one
206:           exists_member([G,L],Pgs), % connection so check it
207:           exists_member(Device,L),  % hasn't been assigned already
208:           \+type(G,Ictype)
209:         ))],                        % actual cable not of interest
210:      [assign_cable_type(Device,Cabletype,Cable,C)],
211:      [],
212:      connect_cable(Device,Cable,C)).
213:
214: method(configure_serial_device(C,Device,PG),
215:     configure(Device,PG,C),
216:     [guess_system_devices(C,SysDev),
217:      \+gmember([Device,_],SysDev),
218:      ((configured(Device,PG,C),!);
219:      (guess_type(cable(Device),Type2),
220:      (Type2 = rs232_serial_cable;Type2 = rs422_serial_cable),
221:      needs_ic(Type2,ICtype),
222:      slots_available(C,PG,Devlist,ICtype,N),
223:      N>0))],
224:      [reduce_slots(PG,Device,Devlist,ICtype,1),
225:      guess_serial_devices(C,SerDev),
226:      ((gmember([Device,_],SerDev),!);add_member([Device,_],SerDev))],
```

```
227:       [],
228:       configure_serial_device(C,Device,PG)).
229:
230: method(h_configure_device(C,Device,_IC),
231:      configure(Device,IC,C),
232:      [guess_serial_devices(C,SerDev),
233:       \+gmember([Device,_],SerDev),
234:       ((configured(Device,_IC,C),!)
235:       ;
236:       (guess_type(Device,Type),
237:       ((guess_system_devices(C,SysDev),
238:         gmember([Device,Cable],SysDev),
239:         guess_type(Cable,Type2)
240:        )
241:        ;
242:        guess_type(cable(Device),Type2)
243:       ),
244:       \+ Type2 = rs232_serial_cable,
245:       \+ Type2 = rs422_serial_cable,
246:       needs_ic(Type2,ICtype),
247:       h_slots_available(C,IC,Type,ICtype,N),    % N is variable mode
248:       N>0))],
249:       [reduce_slots(IC,Device,Type,ICtype,1),
250:       guess_system_devices(C,SysDev),
251:       ((gmember([Device,_],SysDev),!);add_member([Device,_],SysDev))],
252:       [],
253:       h_configure_device(C,Device,_IC)).
254:
255:
256: method(h_configure_ic(C,Ch,CC),
257:       configure(Device,[Ch|Devs],C),
258:       [guess_type(cable(Device),Type2),
259:       \+ Type2 = rs232_serial_cable,
260:       \+ Type2 = rs422_serial_cable,
261:        needs_ic(Type2,ICtype),
262:        \+at_ic_per_cc_heur_limit(C),
263:        \+exist_empty_ics(ICtype,C),
264:        h_cc_available(C,CC,ICtype,N),
265:        N>0],
266:        [add_ic([Ch|Devs],ICtype,CC,C)],
267:       [configure(Device,[Ch|Devs],C)],
268:        h_configure_ic(C,Ch,CC)).
269:
270: method(configure_cc(C,CC),
271:        configure(Dev,Ch,C),
272:        [guess_type(cable(Dev),Type2),
273:         \+ Type2 = rs232_serial_cable,
274:         \+ Type2 = rs422_serial_cable,
275:         (guess_type(Dev,fdisk),\+(exist_empty_ics(fchannel,C))
276:         ;
277:         (\+guess_type(Dev,fdisk),\+(exist_empty_ics(channel,C)))),
278:         \+at_cc_limit(C)],
```

```
279:        [guess_cardcages(C,CCs),
280:        guess_type(CC,cardcage),
281:        add_device(CC,CCs),
282:        add_cc(CC,C)],
283:        [configure(Dev,Ch,C)],
284:        configure_cc(C,CC)).
285:
286: method(configure_portgroups(C,G,Dtc),
287:        configure(Dev,G,C),
288:         [((guess_serial_devices(C,SerDev),
289:            exists_member([Dev,Cable],SerDev),
290:            \+configured(Dev,_,C),
291:            guess_type(Cable,Type2),!)
292:          ;
293:          guess_type(cable(Dev),Type2)),
294:         (Type2 = rs232_serial_cable;Type2 = rs422_serial_cable),
295:          guess_type(IC,lanic),
296:          needs_ic(Type2,Type),
297:          guess_type(G,Type),
298:          guess_lanics(C,Ls),
299:          \+exist_empty_portgroups(C),
300:          guess_portgroups(C,Pgs),
301:          exists_member([IC,Dtclist],Ls),
302:          exists_member([Dtc,PGS],Dtclist),
303:          slots_available([Dtc,PGS],Type,N),
304:         N>0],
305:        [add_device(G,PGS),add_device([G,_],Pgs)],
306:        [configure(Dev,G,C)],
307:        configure_portgroups(C,G,Dtc)).
308:
309: method(configure_serial_connection(C,Dtc,IC),
310:         configure(Device,_,C),
311:         [guess_type(cable(Device),Type2),
312:          \+exist_empty_serial_connections(C),
313:          (Type2 = rs232_serial_cable;Type2 = rs422_serial_cable),
314:          guess_lanics(C,Ls),
315:          exists_member([IC,Dtclist],Ls),
316:          slots_available(C,IC,Dtclist,N),
317:          N>0],
318:          [guess_type(Dtc,serial_conn),
319:           add_device([Dtc,_],Dtclist)],
320:          [configure(Device,_,C)],
321:          configure_serial_connection(C,Dtc,IC)).
322:
323: method(system_disks(C,DevList),
324:      system_disks(C,DevList),
325:      [C=[P|_],groundp(P),      % must know what the processor is
326:       \+configured(system_disks(C))],
327:      [guess_system_disks(C,DevList)],
328:      [repeat(configure(DevList,_,C))],
329:      system_disks(C,DevList)).
330:
```

```
331: method(configure_device_list(C,DevList),
332:       G,
333:       [G=..[Dev,C,DevList],
334:       Term=..[Dev,C],
335:       device(Dev),
336:       \+configured(Term)],
337:       [run_goal(G,DevList1),
338:       DevList = DevList1],
339:       [repeat(configure(DevList1,_,C))],
340:       configure_device_list(C,DevList1)).
341:
342: method(disk_storage(C,Cap,Dks),
343:       disk_capacity(C,Cap),
344:       [C=[P|_],groundp(P),    % must know what the processor is
345:        guess_disks(C,Dks),
346:        (var(Dks);(capacity(Dks,Cap1,Cap1<Cap))],
347:       [generate_disks(C,Dks,Cap)],
348:       [repeat(configure(Dks,_,C))],
349:       disk_storage(C,Cap,Dks)).
350:
351: method(total_disk_storage(C,Cap,Sys,Dks),
352:       total_disk_capacity(C,Cap),
353:       [C=[P|_],groundp(P),      % must know what the processor is
354:        guess_disks(C,Dks),
355:        var(Dks)],               % Note system disks could be ground
356:       [guess_system_disks(C,Sys),
357:        generate_all_disks(C,Sys,Dks,Cap)],
358:       [repeat(configure(Dks,_,C)),repeat(configure(Sys,_,C))],
359:       total_disk_storage(C,Cap,Sys,Dks)).
360:
361: method(tape_capacity(C,Cap,Tps),
362:       backup_capacity(C,Cap),
363:       [guess_tapes(C,Tps),
364:        var(Tps)],                 % can weaken (see text)
365:       [generate_tapes(C,Tps,Cap)],
366:       [repeat(configure(Tps,_,C))],
367:       tape_capacity(C,Cap,Tps)).
368:
369: method(is_legal(C),
370:       legal(C),
371:       [guess_system_disks(C,Sys),
372:        groundp(Sys),
373:        guess_tapes(C,B),
374:        groundp(B),
375:        check_not_share_constraints(C)],
376:        [],
377:        [],
378:        is_legal(C)).
379:
380: method(add_system_disks(C,Sys),
381:       legal(C),
382:       [C=[P|_],groundp(P),       % must know what the processor is
```

```
383:       guess_system_disks(C,Sys),
384:       var(Sys)],
385:       [generate_system_disks(C,Sys)],
386:       [repeat(configure(Sys,_,C)),legal(C)],
387:       add_system_disks(C,Sys)).
388:
389: method(add_backup(C,Tps),
390:      legal(C),
391:       [guess_tapes(C,Tps),
392:       var(Tps)],
393:       [generate_tapes(C,Tps)],
394:       [repeat(configure(Tps,_,C)),legal(C)],
395:       add_backup(C,Tps)).
396:
397: method(configure_device(C,Device,_IC),
398:     configure(Device,IC,C),
399:     [guess_serial_devices(C,SerDev),
400:      \+gmember([Device,_],SerDev),
401:      ((configured(Device,_IC,C),!)
402:      ;
403:      (guess_type(Device,Type),
404:      ((guess_system_devices(C,SysDev),
405:        gmember([Device,Cable],SysDev),
406:        guess_type(Cable,Type2)
407:       )
408:       ;
409:       guess_type(cable(Device),Type2)
410:      ),
411:      \+ Type2 = rs232_serial_cable,
412:      \+ Type2 = rs422_serial_cable,
413:      needs_ic(Type2,ICtype),
414:      slots_available(C,IC,Type,ICtype,N),    % N is variable mode
415:      N>0))],
416:      [reduce_slots(IC,Device,Type,ICtype,1),
417:      guess_system_devices(C,SysDev),
418:      ((gmember([Device,_],SysDev),!);add_member([Device,_],SysDev))],
419:      [],
420:      configure_device(C,Device,_IC)).
421:
422: method(configure_ic(C,Ch,CC),
423:      configure(Device,[Ch|Devs],C),
424:      [guess_type(cable(Device),Type2),
425:      \+ Type2 = rs232_serial_cable,
426:      \+ Type2 = rs422_serial_cable,
427:       needs_ic(Type2,ICtype),
428:       \+exist_empty_ics(ICtype,C),
429:       cc_available(C,CC,ICtype,N),
430:       N>0],
431:       [add_ic([Ch|Devs],ICtype,CC,C)],
432:       [configure(Device,[Ch|Devs],C)],
433:       configure_ic(C,Ch,CC)).
434:
```

```
435: method(configure_extra_ic(C,Ch,_CC),
436:        legal(C),
437:       [\+at_ic_per_cc_heur_limit(C),
438:        \+exist_empty_ics(channel,C),
439:        cc_available(C,CC,channel,N),
440:        N>0],
441:        [add_ic([Ch|_Devs],channel,CC,C)],
442:        [legal(C)],
443:        configure_ic(C,Ch,_CC)).
444:
445:
446: submethod(configure_disks(C),
447:               Inputconj,
448:               [[system_disks(C,Sys),disk_capacity(C,DCap),
449:                 disks(C,Dks),total_disk_capacity(C,TDCap)]],
450:               [],
451:               [(applicable_subm(Inputconj,
452:                           method(system_disks(C,Sys),_,_,_,_,_,Tac1),
453:                           Outconj1)
454:                 ;
455:                 (Tac1=[],Outconj1=Inputconj)),
456:
457:                 (applicable_subm(Outconj1,
458:                                  method(configure_device_list(C,Dks),
459:                                         disks(C,Dks),_,_,_,_,Tac2),
460:                                  Outconj2)
461:                 ;
462:                 (Tac2=[],Outconj2=Outconj1)
463:                 ),
464:                 (applicable_subm(Outconj2,
465:                        method(total_disk_storage(C,TDCap,Sys,Dks),
466:                                                  _,_,_,_,_,Tac3),
467:                                  Outconj3)
468:                 ;
469:                 (Tac3=[],Outconj3=Outconj2)),
470:                  (applicable_subm(Outconj3,
471:                                  method(disk_storage(C,DCap,Dks),
472:                                         _,_,_,_,_,Tac4),
473:                                  Outputconj)
474:                  ;
475:                 (Tac4=[],Outputconj=Outconj3)),
476:                 flatten([Tac1,Tac2,Tac3,Tac4],Tactic)
477:               ],
478:               Outputconj,
479:               Tactic).
```

D.3 Planning

D.3.1 The planner

```
%%%
```

```
47:      plantraced(2,
48:                (tab(57),write('... '),
49:                 write(' applicable'),nl,nl)),
50:      plantraced(1,
51:        (pprint(Method),nl,write(' on goal '),
52:         pprint(Input),nl,nl)),
53:    append(Plansofar,Tactic,Extendedplan),% Tactic is list (for now)
54:    find_plan(Outputconj,Extendedplan,Plan).
55:
56: find_plan(Inputconj,Plansofar,Plan):-
57:      method(Method,Input,Preconditions,Effects,Output,Tactic),
58:      \+Method = is_legal(_),
59:      member(Input,Inputconj),
60:      plantraced(2,(nl,write('Trying '),pprint(Method),
61:           nl,write(' on goal '),pprint(Input),write(' ... '),nl,nl)),
62:      applicable(method(Method,Input,Preconditions,Effects,
63:                                                    Output,Tactic)),
64:      plantraced(2,
65:                (tab(57),write('... '),
66:                 write(' applicable'),nl,nl)),
67:      plantraced(1,
68:        (pprint(Method),nl,write(' on goal '),
69:         pprint(Input),nl,nl)),
70:      append(Plansofar,[Tactic],Extendedplan),
71:      replace(Input,Output,Inputconj,Outputconj),
72:      find_plan(Outputconj,Extendedplan,Plan).
73:
74: applicable(method(_,_,Preconditions,Effects,_,_)):-
75:      test_conditions(Preconditions),
76:      apply_effects(Effects).
77:
78: applicable(supermethod(_,_,_,Preconditions,Effects,_,_)):-
79:      test_conditions(Preconditions),
80:      apply_effects(Effects).
81:
82: applicable_subm(Inputconj,
83:                method(Name,Input,Pre,Eff,Out,Tactic),
84:                Outputconj):-
85:                          method(Name,Input,Pre,Eff,Out,Tactic),
86:                          member(Input,Inputconj),
87:                          test_conditions(Pre),
88:                          apply_effects(Eff),
89:                          replace(Input,Out,Inputconj,Outputconj).
90:
91: applicable_subm(Inputconj,
92:                supermethod(Name,Inputconj,Input,Pre,Eff,_Out,Tactic),
93:                Outputconj):-
94:             supermethod(Name,Inputconj,Input,Pre,Eff,Outputconj,Tactic),
95:             subset(Input,Inputconj,_Intersection),
96:             test_conditions(Pre),
97:             apply_effects(Eff).
98:
```

```
 99: applicable_subm(Inputconj,
100:                submethod(Name,Inputconj,Input,Pre,Eff,_Out,Tactic),
101:                Outputconj):-
102:            submethod(Name,Inputconj,Input,Pre,Eff,Outputconj,Tactic),
103:            subset(Input,Inputconj,_Intersection),
104:            test_conditions(Pre),
105:            apply_effects(Eff).
106:
107:
108: test_conditions([]).
109: test_conditions([H|T]):-
110:   plantraced(3,(nl,write('...testing precondition '),write(H),nl)),
111:   plantraced(3,(nl,write('...testing precondition '),pprint(H),nl)),
112:   call(H),
113:   test_conditions(T).
114:
115: apply_effects([]).
116: apply_effects([Effect|Effects]):-
117:    call(Effect),
118:    apply_effects(Effects).
119:
120: execute(Plan):-
121:     nl,
122:     write('EXECUTING'),
123:     nl,
124:     write('---------'),
125:     nl,
126:     execute(Plan,C),
127:     nl,
128:     write('HERE IS THE SYNTHESIZED CONFIGURATION'),
129:     nl,
130:     write('-----------------------------------'),
131:     nl,
132:     nl,
133:     print_system(C).
134:
135: execute([],_).
136: execute([H|T],C):-
137:     alter(H,C,H1),
138:     functor(H1,F,_),
139:     plantraced(2,write(F)),
140:     plantraced(1,(pprint(H1),nl)),
141:     H1,
142:     plantraced(1,(tab(10),pprint(H1),nl)),
143:     execute(T,C).
```

Appendix E

Testing procedure

E.1 Tests

The following is a set of specifications used to benchmark CLEM. I devised 40 tests, based on examples given to novice sales representatives. I evaluated the results by comparing them with what I had learned were acceptable configurations. Unfortunately more extensive field testing was not possible.

```
% START OF SYSTEM DEVICES ONLY TESTING

% Processor specified: no other devices.
% A 'minimum' configuration of one system disk drive
% and one tape drive will be provided.
% They should be configured one on each of the hpib channels
% (not on the same one due to constraints)
% On backtracking, various possibilities should be found:
% For the 7937FL an hpfl channel will be needed.
find(spec1,spec(C)=[processors(C,['950'(_)])]).

% Now specify a specific system disk drive.
% First a hpib-connected drive:
find(spec2,spec(C)=[processors(C,['950'(_)]),system_disks(C,['7937H'(_)])]).
% ... then an hpfl-connected drive
find(spec3,spec(C)=[processors(C,['950'(_)]),system_disks(C,['7937FL'(_)])]).
```

```
% Now specify a specific user disk drive.
%(Two cases, hpib and hpfl)
find(spec4,spec(C)=[processors(C,['950'(_)]),disks(C,['7937H'(_)])]).
find(spec5,spec(C)=[processors(C,['950'(_)]),disks(C,['7937FL'(_)])]).

% Now specify user disks by capacity: a small capacity ...
find(spec6,spec(C)=[processors(C,['950'(_)]),disk_capacity(C,500)]).
% Larger, needing 10 user disk drives.
find(spec7,spec(C)=[processors(C,['950'(_)]),disk_capacity(C,3500)]).
% Larger still ,needing ???
find(spec8,spec(C)=[processors(C,['950'(_)]),disk_capacity(C,5500)]).

% As above, but specifying a system disk drive: two cases for each -
% hpib and hpfl
find(spec9,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,500),
                    system_disks(C,['7937XP'(_)])]).
find(spec10,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,500),
                    system_disks(C,['7937FL'(_)])]).
find(spec11,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,3500),
                    system_disks(C,['7937XP'(_)])]).
find(spec12,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,3500),
                    system_disks(C,['7937FL'(_)])]).
find(spec13,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,5500),
                    system_disks(C,['7937XP'(_)])]).
find(spec14,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,5500),
                    system_disks(C,['7937FL'(_)])]).

% Here the system and user disk capacities are not specified separately.
find(spec15,spec(C)=[processors(C,['950'(_)]),total_disk_capacity(C,1000)]).
% A larger example:
find(spec16,spec(C)=[processors(C,['950'(_)]),total_disk_capacity(C,6000)]).

% Now cases involving tapes explicity specified:
find(spec17,spec(C)=
              [processors(C,['950'(_)]),
               disk_capacity(C,5500),
               system_disks(C,['7937FL'(_)]),
               tapes(C,
                     ['7980XC'(_),'7980XC'(_),'7980XC'(_),'7980XC'(_)])]).

% Now with capacity of tapes specified:
find(spec18,spec(C)=[processors(C,['950'(_)]),disk_capacity(C,5500),
                    backup_capacity(C,5000)]).
find(spec19,spec(C)=[processors(C,['950'(_)]),
                    system_disks(C,['7937XP'(_)]),
                    disk_capacity(C,5500),
```

```
                    backup_capacity(C,5000)]).
find(spec20,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,5500),
                    system_disks(C,['7937FL'(_)]),
                    backup_capacity(C,5000)]).

% An example in which it is impossible to keep to heuristic constraints.
find(spec21,spec(C)=[processors(C,['950'(_)]),disk_capacity(C,16000)]).

% END OF SYSTEM DEVICES ONLY TESTING
```

```
% START OF SERIAL DEVICES ONLY TESTING (except for system disk and tape)

% Needs three portgroups
find(spec22,spec(C)=
          [processors(C,['950'(_)]),
           terminals(C,['2392A'(_),'2392A'(_),'2392A'(_),'2392A'(_),
                        '2392A'(_),'2392A'(_),'2392A'(_),'2392A'(_),
                        '2392A'(_),'2392A'(_),'2392A'(_),'2392A'(_),
                        '2392A'(_)])]).
% Now specify a non-default cable: one terminal ...
find(spec23,spec(C)=[processors(C,['950'(_)]),terminals(C,['2392A'(N)]),
                     cabletype('2392A'(N),rs422_serial_cable,C)]).
% ... and several
find(spec24,spec(C)=[processors(C,['950'(_)]),
                     cabletype('2392A'(N1),rs422_serial_cable,C),
                     cabletype('2392A'(N2),rs422_serial_cable,C),
                     terminals(C,['2392A'(N1),'2392A'(N2),'2394A'(_)])]).


% Configure a printer as a serial device by specifying a serial cable
find(spec25,spec(C)=[processors(C,['950'(_)]),
                     printers(C,[P]),
                     cabletype(P,rs232_serial_cable,C)]).

% Mixtures of terminals and serial printers
find(spec26,spec(C)=[processors(C,['950'(_)]),
                     printers(C,[P]),
                     cabletype(P,rs232_serial_cable,C),
                     terminals(C,[_T1,_T2])]).

% Specify the printer implicitly by various attributes
find(spec27,spec(C)=[processors(C,['950'(_)]),
                     printers(C,[P]),
                     ppm(C,P,10)]).

find(spec28,spec(C)=[processors(C,['950'(_)]),
                     printers(C,[P]),
                     printer_type(C,P,page)]).

find(spec29,spec(C)=[processors(C,['950'(_)]),
                     printers(C,[P]),
                     printer_type(C,P,page),
                     ppm(C,P,10)]).

% END OF SERIAL DEVICES ONLY TESTING

% MIXTURES OF SYSTEM AND SERIAL DEVICES

% 1 user disk, 1 system disk, 1 serial printer
find(spec30,spec(C)=[processors(C,['950'(_)]),
                     disk_capacity(C,500),
                     lpm(C,P1,500),
```

```
                    printers(C,[P1]),
                    cabletype(P1,rs232_serial_cable,C),
                    system_disks(C,['7937FL'(_)]),
                    tapes(C,['7980XC'(_)])]).
find(spec31,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,500),
                    lpm(C,P1,600),
                    cabletype(P1,rs232_serial_cable,C),
                    printers(C,[P1,_P2])]). % additional printer

% Complete specifications, to be configured using basic_config

find(spec32,spec(C)=
            [processors(C,['950'(_)]),disk_capacity(C,1670),
             terminals(C,[_T1,_T2,_T3,_T4,_T5,_T6,_T7,_T8,_T9,_T10,
                          _T11,_T12,_T13,_T14,_T15,_T16]),
             backup_capacity(C,296),printers(C,[P1,P2]),lpm(C,P1,600),
             cps(C,P2,300)]).

find(spec33,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,1670),
                    terminals(C,[_T1,_T2]),
                    printers(C,[_P1,_P2])]).
find(spec34,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,1670),
     system_disks(C,['7937XP'(_)]),
                    terminals(C,[_T1,_T2]),
                    printers(C,[_P1,_P2])]).
find(spec35,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,1670),
     system_disks(C,['7937XP'(_)]),
                    terminals(C,['2392A'(_),'2392A'(_)]),
                    printers(C,['2235A'(_),'2932A'(_)])]).
find(spec36,spec(C)=[processors(C,['950'(_)]),
                    disk_capacity(C,1670),
                    lpm(C,P1,600),
     system_disks(C,['7937XP'(_)]),
                    terminals(C,[_T1,_T2]),
                    printers(C,[P1,_P2])]).
find(spec37,spec(C)=[processors(C,['950'(_)]),
                    total_disk_capacity(C,200),
                    terminals(C,[T1,_T2]),
             cabletype(T1,rs422_serial_cable,C),
                    printers(C,[P1,P2]),
                    cabletype(P2,system_cable,C),
                    lpm(C,P1,600),
                    cps(C,P2,300)]).
```

```
% Some larger or awkward examples

% Basic config not applicable (no printers)
find(spec38,spec(C)=
               [processors(C,['950'(_)]),
                disk_capacity(C,5500),
                terminals(C,['2392A'(_),'2392A'(_),'2392A'(_),'2392A'(_),
                             '2392A'(_),'2392A'(_),'2392A'(_),'2392A'(_),
                             '2392A'(_),'2392A'(_),'2392A'(_),'2392A'(_),
                             '2394A'(_),'2394A'(_),'2394A'(_),
                             '2394A'(_)]),
                system_disks(C,['7937FL'(_)]),
                tapes(C,
                     ['7980XC'(_),'7980XC'(_),'7980XC'(_),'7980XC'(_)])]).

find(spec39,spec(C)=
                [processors(C,['950'(_)]),
                 disk_capacity(C,4670),
                 terminals(C,[T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,
                              T14,T15,T16]),
                 cabletype(T1,rs422_serial_cable,C),
                 cabletype(T2,rs422_serial_cable,C),
                 cabletype(T3,rs422_serial_cable,C),
                 cabletype(T4,rs422_serial_cable,C),
                 cabletype(T5,rs422_serial_cable,C),
                 cabletype(T6,rs422_serial_cable,C),
                 cabletype(T7,rs422_serial_cable,C),
                 cabletype(T8,rs422_serial_cable,C)
                 cabletype(T9,rs422_serial_cable,C),
                 cabletype(T10,rs422_serial_cable,C),
                 cabletype(T11,rs422_serial_cable,C),
                 cabletype(T12,rs422_serial_cable,C),
                 cabletype(T13,rs422_serial_cable,C),
                 cabletype(T14,rs422_serial_cable,C),
                 cabletype(T15,rs422_serial_cable,C),
                 cabletype(T16,rs422_serial_cable,C),
                 backup_capacity(C,296),printers(C,[P1,P2]),lpm(C,P1,600),
                 cps(C,P2,300)]).

% One which cannot be configured using heuristic limits
find(spec40,spec(C)=
                 [processors(C,['950'(_)]),
                  disk_capacity(C,16000),
                  terminals(C,[T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,
                               T11,T12,T13,T14,T15,T16]),
                  cabletype(T1,rs422_serial_cable,C),
                  cabletype(T2,rs422_serial_cable,C),
                  cabletype(T3,rs422_serial_cable,C),
                  cabletype(T4,rs422_serial_cable,C),
                  cabletype(T5,rs422_serial_cable,C),
                  cabletype(T6,rs422_serial_cable,C),
                  cabletype(T7,rs422_serial_cable,C),
```

```
                    cabletype(T8,rs422_serial_cable,C),
                    cabletype(T9,rs422_serial_cable,C),
                    cabletype(T10,rs422_serial_cable,C),
                    cabletype(T11,rs422_serial_cable,C),
                    cabletype(T12,rs422_serial_cable,C),
                    cabletype(T13,rs422_serial_cable,C),
                    cabletype(T14,rs422_serial_cable,C),
                    cabletype(T15,rs422_serial_cable,C),
                    cabletype(T16,rs422_serial_cable,C),
                    backup_capacity(C,296),
                    printers(C,[P1,P2]),lpm(C,P1,600),
                    cps(C,P2,300)]).

% This one finds a non-executable plan, then replans
% Basic config not applicable (no printers)
find(spec41,spec(C)=[processors(C,['950'(_)]),
                     printers(C,['2680A'(_),'2680A'(_),'2680A'(_)]),
                     system_disks(C,['7937XP'(_)]),
                     tapes(C,['7980XC'(_),'7980XC'(_)])]).

% This one finds a non-executable plan: doesn't replan/terminate
% in reasonable time
find(spec42,spec(C)=
                  [processors(C,['950'(_)]),
                   disk_capacity(C,5500),
                   terminals(C,['2392A'(_),'2392A'(_),'2392A'(_),
                                '2392A'(_),'2392A'(_),'2392A'(_),
                                '2392A'(_),'2392A'(_),'2392A'(_),
                                '2392A'(_),'2392A'(_),'2392A'(_),
                                '2394A'(_),'2394A'(_),'2394A'(_),
                                '2394A'(_)]),
                   printers(C,
                            ['2680A'(_),'2680A'(_),'2680A'(_),'2680A'(_)]),
                   system_disks(C,['7937XP'(_)]),
                   tapes(C,
                         ['7980XC'(_),'7980XC'(_),'7980XC'(_)])]).
```

E.2 Plans which fail

This is an example of weak preconditions allowing a plan to be formed which cannot be executed.

```
spec41

-----

specification
     processors(_348830,[950(_348839)])
     printers(_348830,[2680A(_348848),2680A(_348852),2680A(_348856)])
     system_disks(_348830,[7937XP(_348865)])
     tapes(_348830,[7980XC(_348874),7980XC(_348878)])

% Note the printers are incompatible with the disk drive.
% The tape drivers are incompatible with the disk drive.

THIS IS THE PLAN                          % 1st attempt
---------------
configure_processor(S,[950(_6807)])
special(S)
system_disks(S,[7937XP(_6833)])
h_configure_device(S,7937XP(_6833),_9380)
configure_device_list(S,[2680A(_6816),2680A(_6820),2680A(_6824)])
h_configure_device(S,2680A(_6816),_11672)
h_configure_device(S,2680A(_6820),_12880)
h_configure_device(S,2680A(_6824),_14128)
configure_device_list(S,[7980XC(_6842),7980XC(_6846)])
h_configure_device(S,7980XC(_6842),_83189)
h_configure_device(S,7980XC(_6846),_84381)
is_legal(S)


EXECUTING
---------
...
system_disks(S,[7937XP(_6833)])
h_configure_device(S,7937XP(3),hpib(8))       % disk drive on 1st channel
configure_device_list(S,
     [2680A(_6816),2680A(_6820),2680A(_6824)])
h_configure_device(S,2680A(3),hpib(9))        % printer1 on 2nd channel
h_configure_device(S,2680A(4),hpib(9))        % printer2 on 2nd channel
h_configure_device(S,2680A(5),hpib(9))        % printer3 on 2nd channel
configure_device_list(S,[7980XC(3),7980XC(4)])
h_configure_device(S,7980XC(3),hpib(9))       % tape1 on 2nd channel
      h_configure_device(S,7980XC(4),_109081) % nowhere for this one to go


THIS IS THE PLAN                          6th attempt
```

```
---------------
configure_processor(S,[950(_8253)])
special(S)
system_disks(S,[7937XP(_8279)])
h_configure_device(S,7937XP(_8279),_10826)
configure_device_list(S,[2680A(_8262),2680A(_8266),2680A(_8270)])
h_configure_device(S,2680A(_8262),_13118)
h_configure_device(S,2680A(_8266),_14326)
h_configure_device(S,2680A(_8270),_15574)
configure_device_list(S,[7980XC(_8288),7980XC(_8292)])
h_configure_device(S,7980XC(_8288),_84601)
h_configure_ic(S,hpib(_9474),950-cardcage(_9420)) % configures an extra ic
h_configure_device(S,7980XC(_8292),_87119)
is_legal(S)


EXECUTING
---------
...
system_disks(S,[7937XP(135)])
h_configure_device(S,7937XP(135),hpib(208))
configure_device_list(S,[2680A(164),2680A(165),2680A(166)])
h_configure_device(S,2680A(164),hpib(209))
h_configure_device(S,2680A(165),hpib(209))
h_configure_device(S,2680A(166),hpib(209))
configure_device_list(S,[7980XC(153),7980XC(154)])
h_configure_device(S,7980XC(153),hpib(209))
h_configure_ic(S,hpib(210),950-cardcage(172))
h_configure_device(S,7980XC(154),hpib(210))
is_legal(S)

HERE IS THE SYNTHESIZED CONFIGURATION
-------------------------------------


processor:     950(9)

console:

system disks:
     7937XP(135) with cable hpib-cable(72)

user disks:

tape drives:
     7980XC(153) with cable hpib-cable(76)
     7980XC(154) with cable hpib-cable(77)

printers:
     2680A(164) with cable hpib-cable(73)
     2680A(165) with cable hpib-cable(74)
     2680A(166) with cable hpib-cable(75)
```

```
terminals:

channel interfaces:
          hpib(208)          with devices
                                        7937XP(135)
          hpib(209)          with devices
                                        7980XC(153)
                                        2680A(164)
                                        2680A(165)
                                        2680A(166)
          hpib(210)          with devices
                                        7980XC(154)


serial connections:
     lnc(136) with DTCs:

Devices connected via serial ports as follows:

card cages:
          950-cardcage(172)  with interfaces
                                        mx(144)
                                        mx(145)
                                        lnc(136)
                                        hpib(208)
                                        hpib(210)
          950-cardcage(173)  with interfaces
                                        hpib(209)


yes
| ?-
```

However, notice that the alternative version of the planner, with stronger preconditions, immediately finds an executable plan:

```
THIS IS THE PLAN
----------------
configure_processor(S,[950(_6805)])
special(S)
system_disks(S,[7937XP(_6831)])
h_configure_device(S,7937XP(_6831),_7748)
configure_device_list(S,[2680A(_6814),2680A(_6818),2680A(_6822)])
h_configure_device(S,2680A(_6814),_8658)
h_configure_device(S,2680A(_6818),_9193)
h_configure_device(S,2680A(_6822),_9743)
configure_device_list(S,[7980XC(_6840),7980XC(_6844)])
h_configure_device(S,7980XC(_6840),_10671)
h_configure_ic(S,hpib(_11464),950-cardcage(_7220))
h_configure_device(S,7980XC(_6844),_11731)
```

```
is_legal(S)


EXECUTING
---------

HERE IS THE SYNTHESIZED CONFIGURATION
-------------------------------------


processor:     950(47)

console:

system disks:
     7937XP(238) with cable hpib-cable(530)

user disks:

tape drives:
     7980XC(7) with cable hpib-cable(534)
     7980XC(8) with cable hpib-cable(535)

printers:
     2680A(1222) with cable hpib-cable(531)
     2680A(1223) with cable hpib-cable(532)
     2680A(1224) with cable hpib-cable(533)

terminals:

channel interfaces:
          hpib(176)          with devices
                                        7937XP(238)
          hpib(177)          with devices
                                        7980XC(7)
                                        2680A(1222)
                                        2680A(1223)
                                        2680A(1224)
          hpib(178)          with devices
                                        7980XC(8)


serial connections:
     lnc(47) with DTCs:

Devices connected via serial ports as follows:

card cages:
          950-cardcage(14360)  with interfaces
                                        mx(93)
                                        mx(94)
                                        lnc(47)
```

```
                                        hpib(176)
                                        hpib(178)
        950-cardcage(14361)  with interfaces
                                        hpib(177)


yes
```

A similar problem exists for the next example, on a bigger scale.

```
spec42

-----

specification
     processors(_356563,[950(_356572)])
     disk_capacity(_356563,5500)
     terminals(_356563,[2392A(_356586),2392A(_356590),...
     printers(_356563,
          [2680A(_356655),2680A(_356659),2680A(_356663),2680A(_356667)])
     system_disks(_356563,[7937XP(_356676)])
     tapes(_356563,[7980XC(_356685),7980XC(_356689),7980XC(_356693)])



THIS IS THE PLAN
----------------
configure_processor(S,[950(_356572)])
special(S)
system_disks(S,[7937XP(_356676)])
disk_storage(S,5500,[7937XP(_358308), (10 disk drives)
configure_device_list(S,[7980XC(_356685),7980XC(_356689),7980XC(_356693)])
h_configure_device(S,7937XP(_358308),_492163)
h_configure_device(S,7937XP(_358316),_492645)
h_configure_device(S,7937XP(_358320),_493129)
h_configure_device(S,7937XP(_358324),_493615)
h_configure_device(S,7937XP(_358328),_494103)
h_configure_device(S,7937XP(_358332),_494593)
h_configure_device(S,7980XC(_356685),_495085)
h_configure_device(S,7980XC(_356689),_495593)
h_configure_ic(S,hpib(_496587),950-cardcage(_357338))
h_configure_device(S,7937XP(_358336),_496647)
h_configure_device(S,7937XP(_358340),_497147)
h_configure_device(S,7937XP(_358344),_497649)
h_configure_device(S,7980XC(_356693),_498153)
configure_cc(S,950-cardcage(_499040))
h_configure_ic(S,hpib(_499596),950-cardcage(_499040))
h_configure_device(S,7937XP(_358348),_499678)
h_configure_device(S,7937XP(_356676),_500190)
configure_device_list(S,
            [2680A(_356655),2680A(_356659),2680A(_356663),2680A(_356667)])
```

```
configure_device_list(S,(terminals)
h_configure_device(S,2680A(_356655),_501750)
h_configure_device(S,2680A(_356659),_502363)
h_configure_ic(S,hpib(_503488),950-cardcage(_357342))
h_configure_device(S,2680A(_356663),_503641)
h_configure_device(S,2680A(_356667),_504260)
configure_serial_connection(S,dtc(_505319),lnc(_357368))
(rest of serial device configuration)
is_legal(S)


EXECUTING
---------

Replans: after some time still has not achieved an executable plan.
```

However the strong precondition version has no difficulty:

```
THIS IS THE PLAN
----------------
configure_processor(S,[950(_6910)])
special(S)
system_disks(S,[7937XP(_7054)])
h_configure_device(S,7937XP(_7054),_8011)
disk_storage(S,5500,[7937XP(_6919), (10 disk drives)
h_configure_device(S,7937H(_6919),_9130)
h_configure_device(S,7937H(_6923),_9665)
h_configure_device(S,7937H(_6927),_10215)
h_configure_device(S,7937H(_6931),_10760)
h_configure_device(S,7937H(_6935),_11320)
h_configure_ic(S,hpib(_12121),950-cardcage(_7463))
h_configure_device(S,7937H(_6939),_12388)
h_configure_device(S,7937H(_6943),_12960)
h_configure_device(S,7937H(_6947),_14073)
configure_cc(S,950-cardcage(_115679))
h_configure_ic(S,hpib(_117121),950-cardcage(_115679))
h_configure_device(S,7937H(_6951),_117756)
h_configure_device(S,7937H(_6955),_119245)
configure_device_list(S,(terminals))
configure_serial_connection(S,dtc(_123272),lnc(_7493))
(rest of serial configuration)
configure_device_list(S,
                [2680A(_7033),2680A(_7037),2680A(_7041),2680A(_7045)])
h_configure_ic(S,hpib(_154422),950-cardcage(_7467))
h_configure_device(S,2680A(_7033),_155054)
h_configure_device(S,2680A(_7037),_156679)
h_configure_device(S,2680A(_7041),_158379)
h_configure_device(S,2680A(_7045),_160154)
configure_device_list(S,[7980XC(_7063),7980XC(_7067),7980XC(_7071)])
h_configure_device(S,7980XC(_7063),_181162)
h_configure_device(S,7980XC(_7067),_182770)
```

```
h_configure_device(S,7980XC(_7071),_184453)
is_legal(S)


EXECUTING
---------

HERE IS THE SYNTHESIZED CONFIGURATION
-------------------------------------


processor:     950(48)

console:

system disks:
     7937XP(239) with cable hpib-cable(536)

user disks:
     7937H(230) with cable hpib-cable(537)
     7937H(231) with cable hpib-cable(538)
     7937H(232) with cable hpib-cable(539)
     7937H(233) with cable hpib-cable(540)
     7937H(234) with cable hpib-cable(541)
     7937H(235) with cable hpib-cable(542)
     7937H(236) with cable hpib-cable(543)
     7937H(237) with cable hpib-cable(544)
     7937H(238) with cable hpib-cable(545)
     7937H(239) with cable hpib-cable(546)

tape drives:
     7980XC(9) with cable hpib-cable(547)
     7980XC(10) with cable hpib-cable(548)
     7980XC(11) with cable hpib-cable(549)

printers:
     2680A(1225) with cable 40242X(564)
     2680A(1226) with cable 40242X(565)
     2680A(1227) with cable 40242X(566)
     2680A(1228) with cable 40242X(567)

terminals:
     2392A(393) with cable 40242X(548)
          etc.

channel interfaces:
          hpib(179)          with devices
                                        7937XP(239)
                                        7937H(230)
                                        7937H(231)
          hpib(180)          with devices
                                        7937H(232)
```

```
                                        7937H(233)
                                        7937H(234)
                                        7980XC(9)
          hpib(181)          with devices
                                        7937H(235)
                                        7937H(236)
                                        7937H(237)
                                        7980XC(10)
          hpib(182)          with devices
                                        7937H(238)
                                        7937H(239)
                                        7980XC(11)
     hpib(183)    {empty}


serial connections:
     lnc(48) with DTCs:
        dtc(14) connecting port groups:
            rs232ports(840)
            rs232ports(842)
            rs232ports(843)


Devices connected via serial ports as follows:
rs232ports(840)
     2392A(393) 2392A(394) 2392A(395) 2392A(396) 2392A(397)
     2392A(398) 2392A(399) 2392A(400)
rs232ports(842)
     2392A(401) 2392A(402) 2392A(403) 2392A(404) 2394A(1)
     2394A(2)   2394A(3)   2394A(4)
rs232ports(843)
     2680A(1225) 2680A(1226) 2680A(1227) 2680A(1228)

card cages:
          950-cardcage(14374)  with interfaces
                                        mx(95)
                                        mx(96)
                                        lnc(48)
                                        hpib(179)
                                        hpib(181)
          950-cardcage(14375)  with interfaces
                                        hpib(180)
                                        hpib(183)
          950-cardcage(14376)  with interfaces
                                        hpib(182)


yes
[debug]
| ?-
```

E.3 Obtaining multiple solutions

Examples of replanning: significantly different configurations.

First, using CLEM's planning mechanism:

```
| ?- config(spec6).

specification
     processors(_6735,[950(_6744)])
     disk_capacity(_6735,500)



THIS IS THE PLAN
----------------
configure_processor(S,[950(_6744)])
special(S)
disk_storage(S,500,[7937H(_7584)])
h_configure_device(S,7937H(_7584),_7653)
add_system_disks(S,[7937H(_8389)])
h_configure_device(S,7937H(_8389),_8504)
add_backup(S,[7979A(_9159)])
h_configure_device(S,7979A(_9159),_9244)
is_legal(S)


Execute this plan(y/n)?n.

THIS IS THE PLAN
----------------
configure_processor(S,[950(_6773)])
special(S)
disk_storage(S,500,[7937H(_7613)])
h_configure_device(S,7937H(_7613),_7682)
add_system_disks(S,[7937H(_8418)])
h_configure_device(S,7937H(_8418),_8533)
add_backup(S,[7979A(_9188)])
h_configure_device(S,7979A(_9188),_9273)
is_legal(S)


Execute this plan(y/n)?n.

THIS IS THE PLAN
----------------
configure_processor(S,[950(_6773)])
special(S)
disk_storage(S,500,[7937H(_7613)])
h_configure_device(S,7937H(_7613),_7682)
add_system_disks(S,[7937H(_8418)])
```

```
h_configure_device(S,7937H(_8418),_8533)
add_backup(S,[7980A(_9188)])
h_configure_device(S,7980A(_9188),_9273)
is_legal(S)


Execute this plan(y/n)?n.

THIS IS THE PLAN
----------------
configure_processor(S,[950(_6773)])
special(S)
disk_storage(S,500,[7937H(_7613)])
h_configure_device(S,7937H(_7613),_7682)
add_system_disks(S,[7937H(_8418)])
h_configure_device(S,7937H(_8418),_8533)
add_backup(S,[7980XC(_9188)])
h_configure_device(S,7980XC(_9188),_9273)
is_legal(S)


Execute this plan(y/n)?y.

EXECUTING
---------

HERE IS THE SYNTHESIZED CONFIGURATION
-------------------------------------


processor:     950(2)

console:

system disks:
     7937H(4) with cable hpib-cable(5)

user disks:
     7937H(3) with cable hpib-cable(4)

tape drives:
     7980XC(1) with cable hpib-cable(6)

printers:

terminals:

channel interfaces:
          hpib(3)          with devices
                                        7937H(3)
                                        7937H(4)
          hpib(4)          with devices
```

```
                                        7980XC(1)

serial connections:
     lnc(2) with DTCs:

Devices connected via serial ports as follows:

card cages:
          950-cardcage(45)  with interfaces
                                        mx(3)
                                        mx(4)
                                        lnc(2)
                                        hpib(3)
          950-cardcage(46)  with interfaces
                                        hpib(4)

Happy(y/n)?y.

yes
| ?-
```

Now using only the object level:

```
| ?- processors(C,['950'(_)]),disk_capacity(C,500),legal(C),print_system(C).

processor:     950(4)
system disks:
     7937H(435) with cable hpib-cable(11)
user disks:
     7937H(436) with cable hpib-cable(10)
tape drives:
     7980A(193) with cable hpib-cable(12)
channel interfaces:
          hpib(83)          with devices
                                        7937H(436)
                                        7937H(435)
          hpib(84)          with devices
                                        7980A(193)

processor:     950(4)
system disks:
     7937H(435) with cable hpib-cable(11)
user disks:
     7937H(436) with cable hpib-cable(10)
tape drives:
     7980A(193) with cable hpib-cable(14)
channel interfaces:
          hpib(83)          with devices
                                        7937H(436)
                                        7980A(193)
          hpib(84)          with devices
```

```
                                        7937H(435)
processor:     950(4)
system disks:
     7937H(435) with cable hpib-cable(17)
user disks:
     7937H(436) with cable hpib-cable(10)
tape drives:
     7980A(193) with cable hpib-cable(18)
channel interfaces:
          hpib(83)          with devices
                                        7937H(435)
          hpib(84)          with devices
                                        7937H(436)
                                        7980A(193)
```

Appendix F

Statistics

F.1 With and without planning

This compares the total time taken by CLEM with the time taken if the object-level theory is used alone, without benefit of planning. The overhead is soon recuperated for any but the most trivial of specifications.

The tests used are the first few tests given in Appendix E.

Test	Total	Obj-l
1	84	67
2	100	100
3	133	135
4	150	100
5	150	100

Test	Total	Obj-l
6	167	1384
7	651	6517
8	1035	9457
9	184	1542
10	200	1134

F.2 Run times for tests spec1–spec40

This gives the CPU time in milliseconds, for each of the tests given in Appendix E: planing time, execution time, total time, and the ratio of planning to execution time ('gearing').

Test	Plan	Exec	Total	Gearing	Test	Plan	Exec	Total	Gearing
1	17	67	84	3.8	22	700	500	1200	0.7
2	33	67	100	2.0	23	75	228	303	3.0
3	50	83	133	1.7	24	150	400	550	2.7
4	50	100	150	2.0	25	117	237	354	2.0
5	50	100	150	2.0	26	150	267	417	1.8
6	50	117	167	2.3	27	50	217	267	4.3
7	270	381	651	1.4	28	50	217	267	4.3
8	435	600	1035	1.4	29	50	217	267	4.3
9	50	134	184	2.7	30	133	133	266	1.0
10	50	150	200	3.0	31	83	433	516	5.2
11	234	284	518	1.2	32	1294	1408	2712	1.1
12	217	383	600	1.8	33	213	711	924	3.3
13	467	650	1117	1.4	34	210	742	952	3.5
14	440	600	1040	1.4	35	217	1217	1424	5.6
15	50	183	233	3.7	36	212	685	897	3.2
16	522	735	1257	1.4	37	200	617	817	3.1
17	533	885	1418	1.7	38	1480	1015	2495	0.7
18	480	1000	1480	2.1	39	1413	3299	4712	2.3
19	517	938	1455	1.8	40	4362	7230	11592	1.7
20	500	975	1475	1.7	41	10483	600	11083	0.06
21	5171	3583	8754	0.7	42	-	-	-	-

Note that the planning figure for spec41 totals the cost of failed execution and replanning. See Appendix E for the details.

F.3 Strengthening/weakening preconditions

The planning and execution times from the previous table, WP and WE, are repeated, along with the equivalent times for an alternative version of CLEM with stronger preconditions.

- WP: weaker preconditions, planning time.
- WE: weaker preconditions, execution time.
- SP: stronger preconditions, planning time.

- SE: stronger preconditions, execution time.

Test	WP	WE	SP	SE
6	50	117	50	100
11	234	284	516	270
13	467	650	816	645
17	533	885	1127	716
18	480	1000	1026	813
19	517	938	1254	900
39	1413	3299	1819	3200
40	4362	7230	8946	7056
41	10483	600	144	650
42	-	-	2168	1750